Chapter III: Processors

III Processor

III.1 Definition, Role, and Importance of the Processor

The processor, or microprocessor, is the central processing unit (CPU) of a computer system. It constitutes the core element responsible for executing most of the calculations and coordinating the operations required for the system's functionality. Implemented as an integrated circuit containing millions of transistors, it is organized to execute a sequence of instructions stored in memory. These instructions can correspond to arithmetic operations (such as addition or multiplication) or logical operations (such as comparisons and conditional tests).

In both computer and embedded systems, the processor plays a fundamental role, as it executes instructions through a precise cycle involving fetching instructions from memory, decoding them, executing the corresponding operations, and writing the results back to registers or memory. It also coordinates all system components through its control unit, which manages communication between memory, registers, and peripherals. Additionally, the processor continuously handles data flow to ensure coherence and synchronization across all parts of the system.

The importance of the processor lies in its direct impact on overall system performance. Its clock frequency, expressed in gigahertz, determines the number of instructions that can be executed per second, and thus the speed of program execution. Modern processors often integrate multiple cores capable of executing instructions in parallel, which significantly enhances performance, especially for multitasking applications. In embedded systems, energy

efficiency is a critical factor: the processor must deliver adequate computing power while minimizing energy consumption.

Finally, the chosen architecture (for instance, x86, ARM, or RISC-V) greatly influences software compatibility and development environments, making it a decisive element for the scalability and longevity of a system.

Thus, the processor is far more than a simple hardware component — it is truly the **“brain”** of the system, defining its processing capabilities, efficiency, and adaptability to both current and future technological demands.

III.2 Physical Architecture of a Single-Core Processor

A single-core processor is composed of several functional subsystems that work together to ensure the proper execution of instructions. The internal organization of such a processor is generally illustrated by a block diagram, where each component fulfills a specific role.

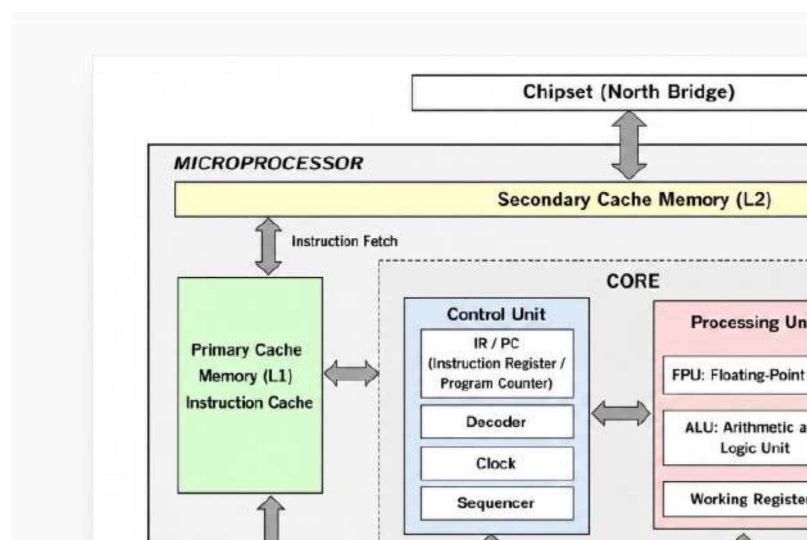


Figure III. 1 - Physical architecture of a single-core processor with its main functional units.

- **Cache Memory:** The processor is directly connected to a primary cache (L1) and a secondary cache (L2). These high-speed memories temporarily store the most frequently used instructions and data to reduce access latency to the main memory. The L1 cache is often divided into two parts: an *instruction cache* and a *data cache*.
- **Processor Core:** The core represents the central part of the processor and includes two main sub-units:
- **Control Unit (CU):** It orchestrates the overall operation of the processor. Its main components include:

- **Decoder**: interprets the instruction and generates the corresponding control signals.
- **Clock**: synchronizes all operations.
- **Sequencer**: manages the orderly execution of the different steps in the instruction cycle.
- **ALU (Arithmetic and Logic Unit):** performs basic arithmetic and logical operations such as addition, subtraction, and comparisons.
- **FPU (Floating Point Unit):** specializes in complex mathematical computations (scientific or graphical).
- **Working Registers:** temporarily store intermediate results.
- **Input/Output Unit:** This unit manages communication between the processor and external components such as main memory, peripherals, and interfaces. It also handles *Direct Memory Access (DMA)* transfers to speed up data exchanges.

III.3 Main Components of the Processor

The processor is composed of several fundamental units that work together to execute instructions. These key elements include the Control Unit (CU), the Arithmetic and Logic Unit (ALU), and a set of registers.

III.3.1 Control Unit (CU)

The Control Unit is responsible for coordinating and managing all internal operations of the processor. It reads instructions from memory, interprets them, and generates the necessary control signals to activate the appropriate functional blocks at the right moment.

The CU consists of several subcomponents:

- The Instruction Register (IR) stores the current instruction being executed.
- The Decoder translates this instruction into a series of micro-operations understandable by the hardware.
- The Sequencer determines the execution order and handles conditional jumps.
- The Clock synchronizes all processor operations to ensure timing accuracy.

Two main types of control units exist:

- Hardwired CU, which is very fast but rigid because it relies on fixed hardware logic.

- Microprogrammed CU, which is more flexible since it uses a memory of micro-instructions but is slightly slower.

In all cases, the Control Unit acts as the true “conductor” of the processor, directing all internal activities in perfect synchrony.

III.3.2 Arithmetic and Logic Unit (ALU)

The Arithmetic and Logic Unit is the computational core of the processor. It performs all fundamental arithmetic operations such as addition, subtraction, multiplication, and division, as well as logical operations such as AND, OR, NOT, and comparisons.

In modern processors, the ALU is often complemented by a Floating Point Unit (FPU), essential for real-number computations encountered in multimedia, scientific, or graphical applications.

From a hardware standpoint, the ALU relies on combinational logic circuits — such as adders, multiplexers, and comparators — which produce results within a very short time, often in a single clock cycle. In addition to the numerical results, the ALU provides status information through flags that indicate specific conditions (e.g., zero result, carry, overflow). These flags are crucial for conditional instructions that depend on computation outcomes.

III.3.3 Processor Registers

Registers are extremely fast internal memory cells located at the heart of the processor. Unlike random-access memory (RAM), they are directly accessible by both the CU and ALU, allowing instantaneous data processing and instruction handling.

Each register plays a specific role within the instruction execution cycle, ranging from storing operands temporarily to managing the address of the next instruction to execute.

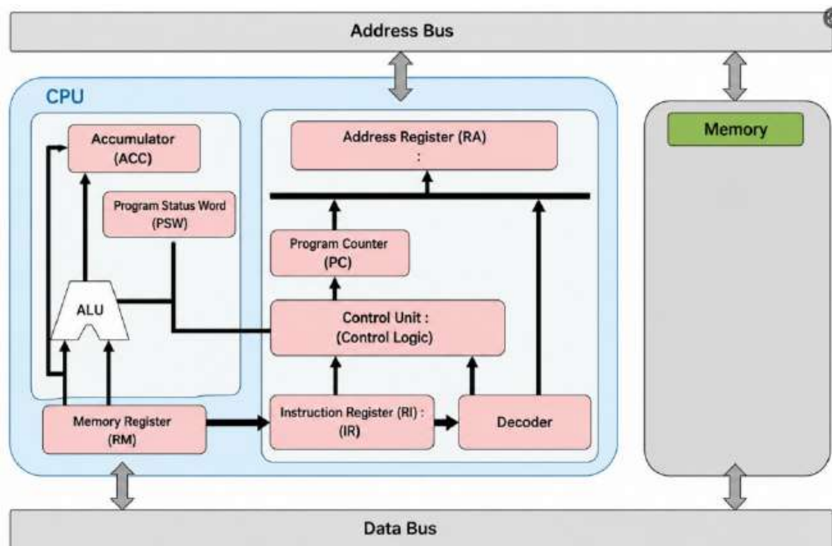


Figure III. 2 - Internal organization of processor registers and their interaction with the ALU, CU, and memory.

The figure above illustrates the main processor registers and their interactions with the ALU, CU, and memory. The following table summarizes the main types of registers and their respective functions:

Tabel III. 1 - Classification of Processor Registers and Their Main Functions

Register Name	Abbreviation	Main Function	Example of Use
Accumulator	ACC	Stores operands and intermediate results from ALU operations.	Result of an addition or comparison.
Status Register	PSW	Contains flags describing the processor's state after an operation.	Indicates whether the result is zero (Zero Flag) or negative (Sign Flag).
Memory Register	MR	Serves as a buffer for data transfers between main memory and the ALU.	Loading a value from RAM for computation.
Address Register	AR	Holds the memory address to be accessed for reading or writing.	Address of a variable in memory.
Program Counter	PC (or CO)	Points to the address of the next instruction to execute.	Sequential transition from one instruction to the next.
Instruction Register	IR	Stores the current instruction before decoding.	Contains "ADD A, B" awaiting execution.
Decoder (linked to IR)	—	Translates the instruction into micro-operations interpretable by the ALU and CU.	Converts "ADD A, B" into internal logic operations.

III.4 Logic and Circuits Used in the ALU

The Arithmetic and Logic Unit (ALU) forms the computational core of the processor, responsible for performing all arithmetic and logical operations. It is built from combinational and sequential circuits composed of basic logic gates. These fundamental building blocks enable complex operations such as addition, subtraction, comparisons, and bit shifts. The accompanying figures illustrate how logic gates operate and how the ALU interacts with the Control Unit (CU) to perform combined operations.

III.4.1 Logic Gates (AND, OR, NOT, XOR)

Logic gates are the fundamental building blocks of all digital circuits. They take binary inputs (0 or 1) and produce a binary output according to a defined logical rule. In the ALU, these gates do not operate independently; they are connected through selection and control mechanisms that determine which operation is executed at a given time. The figure below illustrates this principle:

- Active selection (Selection = 1): the gate is enabled and its result is sent to the output.
- Inactive selection (Selection = 0): the output is placed in a high-impedance (HZ) state, effectively disconnecting it. This prevents conflicts when multiple gates share a common output line.

Thus, although several logic gates are present simultaneously in the ALU, only one gate is activated at a time under the control of the CU through the selection lines.

A) NOT Gate

- Function: Inverts the input (0 becomes 1, 1 becomes 0).
- Operation: When the selection line is active, the output is the logical inversion of the input. If inactive, the output is disconnected (HZ).

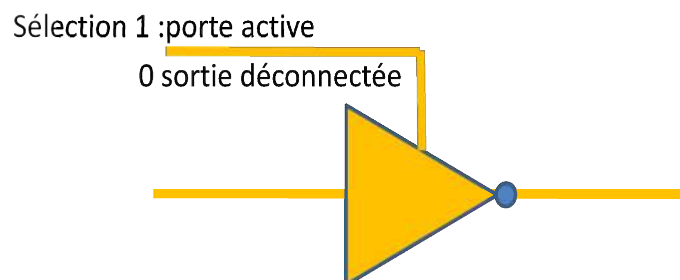


Figure III. 3 - NOT gate with selection line.

B) AND Gate

- Function: Produces 1 only if both inputs are 1.
- The Control Unit activates the AND gate through the selection line; otherwise, its output remains disconnected.

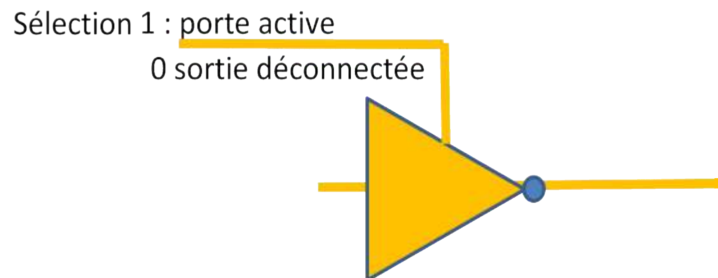


Figure III. 4 - AND gate with selection line.

C) OR Gate

- Function: Produces 1 if at least one input is 1.
- Its activation also depends on the control selection line.

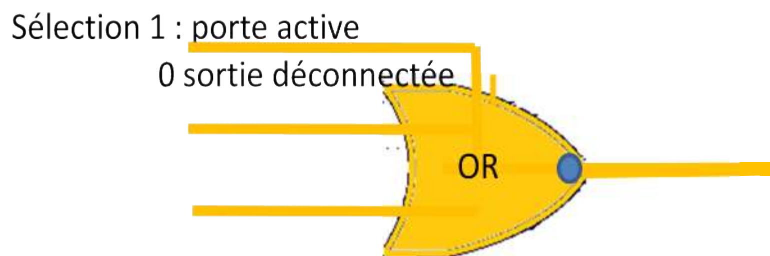


Figure III. 5 - OR gate with selection line.

D) Selection Line

The selection line mechanism demonstrates how the ALU can contain multiple logical operations but execute only one according to the current instruction.

Example:

- For an “AND” instruction → only the AND gate is activated.
- For an “OR” instruction → only the OR gate is activated.
- For a “NOT” instruction → only the NOT gate is activated.

All other gates are placed in **HZ state** to avoid interference on the output bus.

III.4.2 Adders, Multiplexers, and Decoders

From these basic logic gates, more complex circuits can be constructed to perform arithmetic and control operations within the ALU. The three most important circuits are adders, multiplexers (MUX), and decoders.

A) Adders

Adders are the foundation of all arithmetic operations.

- **Half Adder:** Adds two bits (A and B).
  - Sum: $\mathbf{S = A \oplus B}$ (using XOR gate).
  - Carry: $\mathbf{C = A \cdot B}$ (using AND gate).
- **Full Adder:** Adds three bits (A, B, and carry-in, Cin).
  - Sum: $\mathbf{S = A \oplus B \oplus Cin}$
  - Carry-out: $\mathbf{Cout = (A \cdot B) + (Cin \cdot (A \oplus B))}$

In modern processors, faster adders are used, such as the Carry Lookahead Adder (CLA) and the Carry Select Adder (CSA), which minimize carry propagation delays and significantly improve ALU performance.

B) Multiplexers (MUX)

A multiplexer is a switching circuit that selects one input among several and forwards it to a single output, based on control signals.

Example: A 4:1 MUX has four inputs (I0–I3), two selection lines (S0, S1), and one output (Y).

- If S0S1 = 00 → Y = I0
- If S0S1 = 01 → Y = I1
- If S0S1 = 10 → Y = I2
- If S0S1 = 11 → Y = I3

In the ALU, multiplexers are used to:

- Select which operation to perform (e.g., AND, OR, addition, subtraction).
- Choose which operands are applied to the logic gates or adders.

In essence, multiplexers serve as internal routing elements, directing data and commands based on the instruction context.

C) Decoders

Decoders convert a binary combination into a single active output line.

Example: A **3-to-8 decoder** takes 3 input bits and activates one of 8 corresponding output lines.

In a processor, decoders are crucial for:

- Identifying which instruction to execute (e.g., ADD, JUMP, COMPARE).
- Activating only the relevant hardware block within the ALU.
- Generating internal control signals for synchronization and sequencing.

Thus, the decoder serves as a bridge between the Control Unit and the ALU, translating binary instructions into concrete micro-operations for execution.

III.4.3 Sequential Circuits (Flip-Flops and Counters)

Sequential circuits are logic blocks that, unlike combinational circuits, depend not only on the current inputs but also on previous states. They are synchronized by a clock signal that controls timing and enable the storage of binary information. The two fundamental elements of sequential circuits are flip-flops and counters.

A) RS Flip-Flop (Reset–Set)

The **RS flip-flop** is the simplest type. It has two inputs:

- **S (Set):** Forces the output Q to 1.
- **R (Reset):** Forces the output Q to 0.

When S = R = 0, the output retains its previous state. When S = R = 1, the state is undefined and therefore prohibited.

S	R	Q(t+1)	Description
0	0	Q(t)	State preserved
0	1	0	Reset
1	0	1	Set
1	1	Undefined	Forbidden state

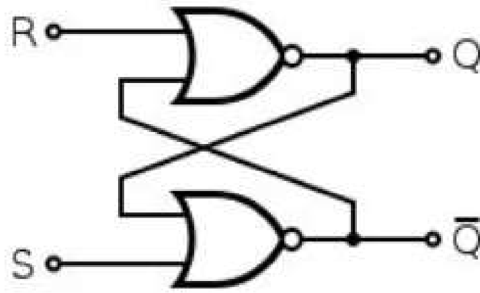


Figure III. 6- Block diagram of the RS flip-flop.

B) D Flip-Flop (Data or Delay)

The D flip-flop eliminates the undefined state of the RS flip-flop and is the most commonly used in modern processors. At every active clock edge, the value present at input **D** is copied to the output **Q**. It is mainly used in the design of **registers** that store data synchronized with the system clock.

D	Clock	Q(t+1)	Description
0	↑	0	Copies a 0 to the output
1	↑	1	Copies a 1 to the output
X	0	Q(t)	No change (idle)

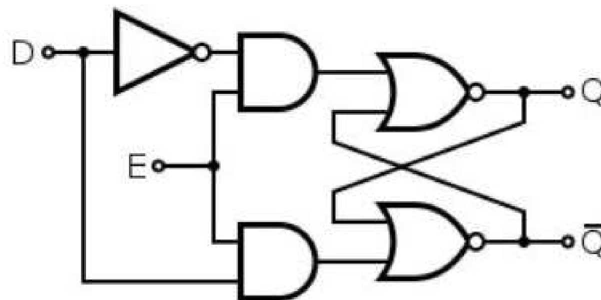


Figure III. 7 - Block diagram of the D flip-flop.

C) JK Flip-Flop

The JK flip-flop is an improved version of the RS type. It removes the forbidden state and adds a toggle feature.

J	K	Q(t+1)	Description
0	0	Q(t)	State preserved
0	1	0	Reset
1	0	1	Set
1	1	¬Q(t)	Toggle (invert)

D) T Flip-Flop (Toggle)

- The **T flip-flop** is derived from the JK type by setting **J = K = 1**.
- It changes (toggles) its state at every active clock edge if **T = 1**.

It is widely used in the design of **binary counters**.

T	Clock	Q(t+1)	Description
0	↑	Q(t)	State preserved
1	↑	¬Q(t)	Toggle (invert)

E) Counters

A counter is a sequential circuit composed of flip-flops (usually T or JK type) that change state according to clock pulses. Each flip-flop represents one bit, and together they form a register capable of storing and updating a binary value. Counters play a fundamental role in sequence generation, time measurement, and synchronization in digital systems.

F) Binary Counter

The simplest form of counter, it increases by one at each clock pulse, cycling through all possible binary states. Its capacity depends on the number of bits (n). For example, a 4-bit counter counts from 0 to 15 (decimal) and then resets to 0. Applications: cyclic binary sequence generation, memory address control, synchronization.

G) Modulo-n Counter

A modulo-n counter counts up to a predefined number of states before resetting to zero (e.g., modulo-10 → counts 0–9). It is used in digital displays, clocks, timers, and systems requiring limited repetitive cycles.

H) Asynchronous Counters (Ripple Counters)

In asynchronous counters, flip-flops do not change state simultaneously. Only the first flip-flop receives the clock signal, while the next ones toggle based on the previous output. This design is simple but introduces **propagation delays**, limiting operating speed.

I) Synchronous Counters

In synchronous counters, all flip-flops receive the same clock pulse and toggle simultaneously.

They are faster and more reliable, making them standard in modern processors and microcontrollers. Counters are used for a variety of purposes:

- Frequency division (to obtain slower clock signals).
- Binary sequence generation for address or data control.
- Timing and delay generation in embedded systems.
- Event counting, period measurement, or speed detection (e.g., in tachometers).

Tabel III. 2 - Comparison of Different Types of Counters

Type of Counter	Operating Principle	Main Characteristics	Typical Applications
Binary Counter	Increments by 1 at each clock pulse, cycling through all binary states.	Simple design, number of states = 2^n (n = number of bits).	Binary sequence generation, memory addressing, automatic cycles.
Modulo-n Counter	Resets to 0 after reaching a predefined maximum value n.	Limited to n states (not necessarily a power of 2).	Digital displays, clocks, repetitive time-based cycles.
Asynchronous Counter (Ripple)	Flip-flops change state sequentially; only the first receives the clock.	Simple, slower due to propagation delays.	Frequency division, low-speed applications.
Synchronous Counter	All flip-flops receive the same clock signal and switch simultaneously.	Fast, reliable, more complex control logic.	Microprocessors, microcontrollers, high-speed embedded systems.

III.5 Operating Principle of the ALU and the Control Unit (CU)

To illustrate how the Arithmetic and Logic Unit (ALU) and the Control Unit (CU) work together, consider the implementation of a logical function defined by the equation:

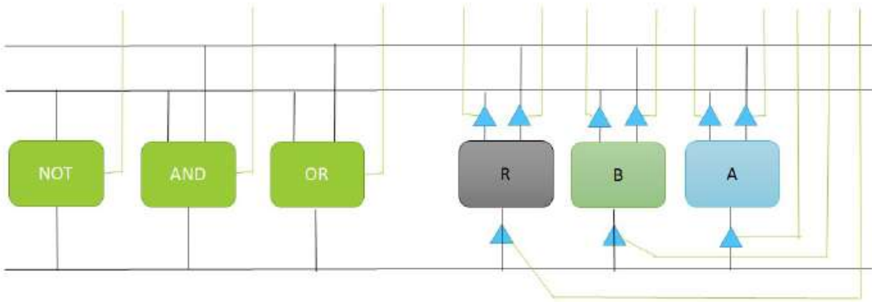
R = Not(A) And Not(B) Or A And B

This function is implemented using three fundamental logic gates: NOT, AND, and OR. The inputs A and B come from processor registers, and the final result is stored in the output register R.

➢ **Step 1 – Circuit Setup**

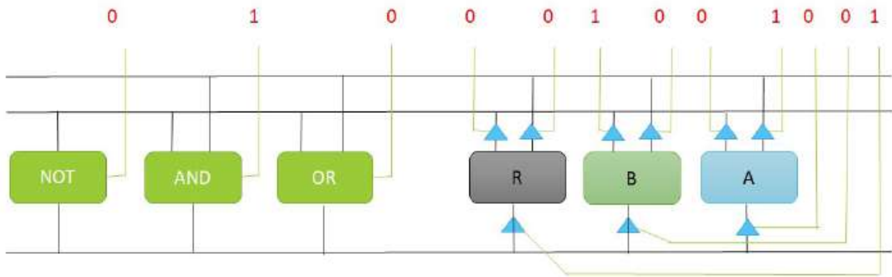
The first diagram shows a simplified architecture.

- Registers **A** and **B** provide binary input values.
- These bits are sent to logic gates as follows:
  - **NOT gates** invert the input signals.
  - **AND gates** perform logical multiplications: (A∧B et Not(A)∧Not(B)).
  - The **OR gate** combines the two partial results to produce the final output **R**.
- The **register R** stores this final result for later use by the processor.



➢ **Step 2 – Applying Binary Data**

Registers **A** and **B** contain sequences of binary values (0 or 1). Each bit is applied simultaneously to the logic circuit. The propagation of signals through the logic gates produces intermediate results—often represented in red in diagrams—allowing the observation of each step in the computation.

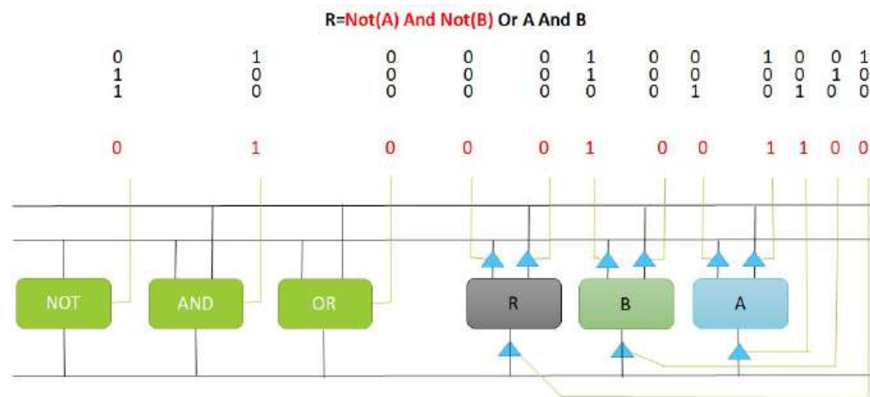


➢ **Step 3 – Intermediate Operations**

The computation proceeds as follows:

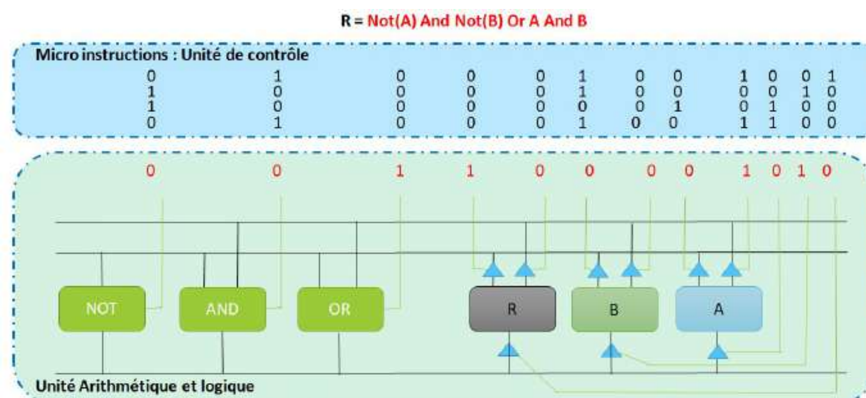
- **Compute (A∧B)** using the first AND gate.
- **Invert A and B** using two NOT gates, producing (\text{NOT}(A)) and (\text{NOT}(B)).
- **Compute (Not(A)∧Not(B))** with a second AND gate.
- **Combine results**: an OR gate merges the two previous results to obtain (**A∧B** and **Not(A)∧Not(B)**).

Each column of the corresponding diagram represents a combination of A and B values, showing the resulting output R.



➢ **Step 4 – Generating and Storing the Output R**

The result **R** is transferred bit by bit into its dedicated register. The stored value can then be reused in subsequent arithmetic or logical operations. This step highlights the crucial role of registers as **temporary working memory** in a processor's architecture.



- **Step 5 – Role of the Control Unit (CU)**

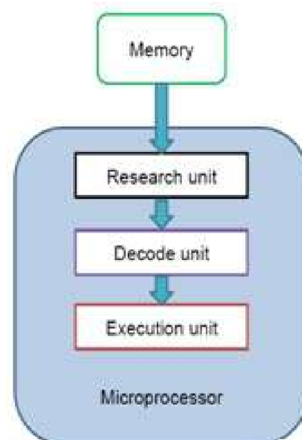
The final diagram introduces **microinstructions** generated by the CU. These instructions specify:

- Which logic gates are to be activated,
- Which data lines are connected,
- How and when the output must be written back into register R.

Thus, the **Control Unit** acts as a **conductor**, organizing the execution process step by step, while the **ALU** performs the actual logical or arithmetic operations.

III.6 Principle of Instruction Execution

The execution of an instruction in a microprocessor follows a strict sequence involving both the Control Unit (CU) and the Arithmetic and Logic Unit (ALU). This process, called the Instruction Cycle, is divided into three main phases: Fetch, Decode, and Execute.



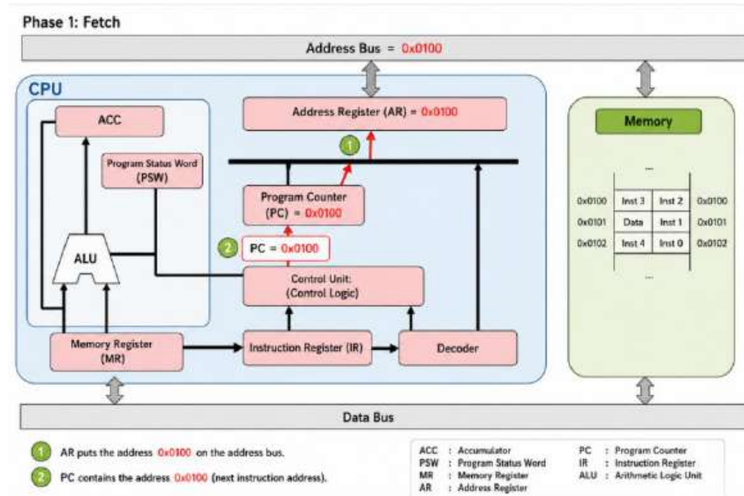
III.6.1 Fetch Phase (Instruction Retrieval)

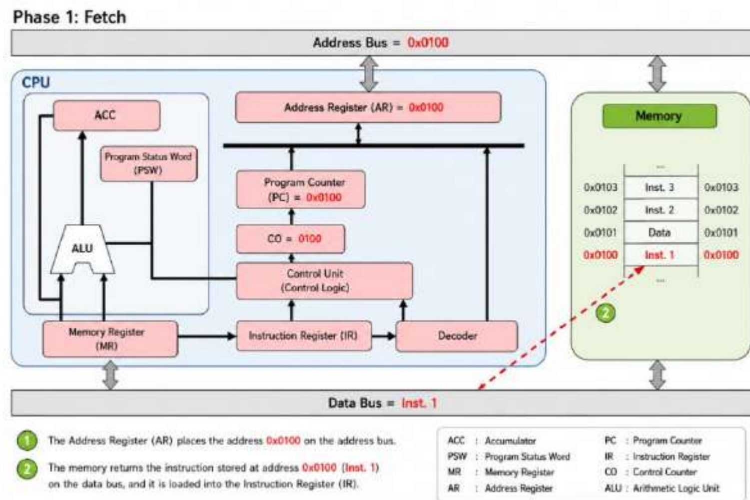
This phase consists of retrieving the instruction to be executed from memory:

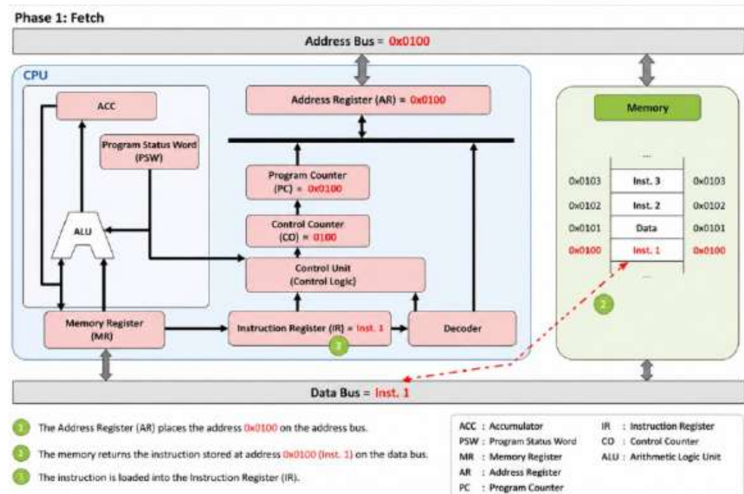
- The Address Register (AR) receives the address of the instruction to load.
- This address is placed on the address bus.
- The memory then sends, via the data bus, the binary code of the instruction (e.g., 0100).
- This instruction is transferred into the Instruction Register (IR).

- In parallel, the Program Counter (PC) is incremented to point to the next instruction.

In the corresponding diagram, address @0x100 is used to fetch instruction Inst_1. The instruction travels from the address bus to memory, then back through the data bus to be stored in the IR.





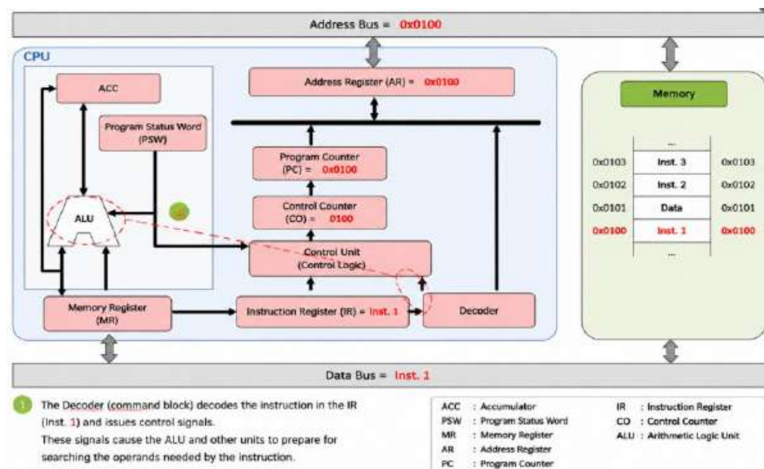


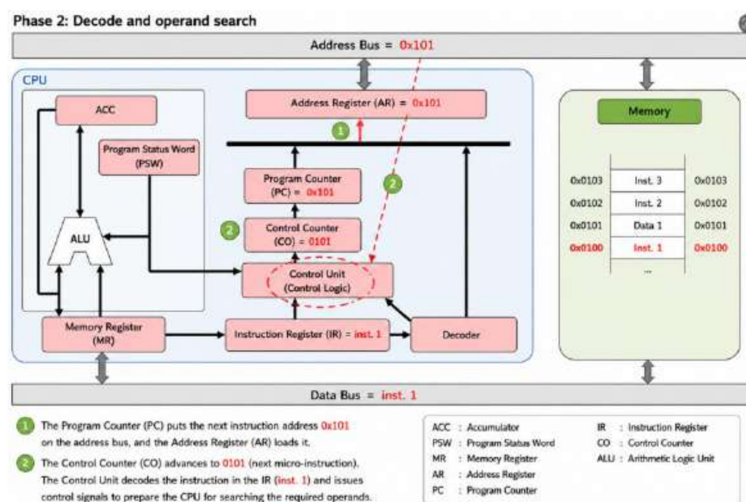
III.6.2 Decode Phase (Instruction Decoding and Operand Retrieval)

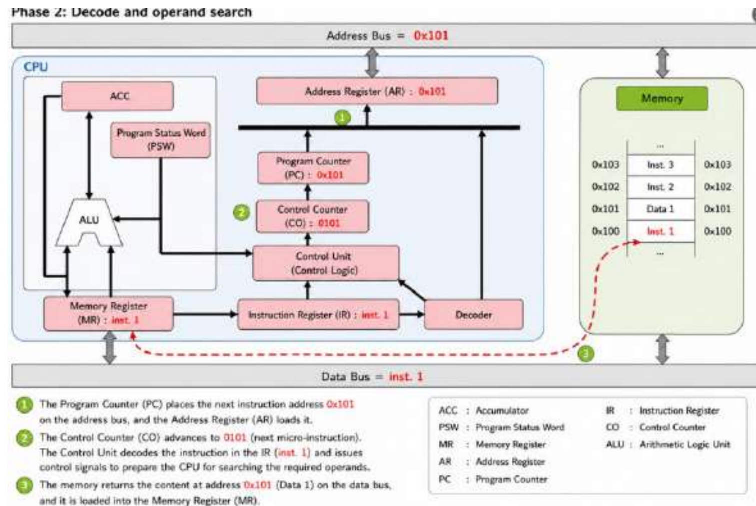
Once the instruction is loaded into the Instruction Register (IR), it is sent to the decoder:

- The Instruction Decoder analyzes the binary code to determine which operation must be performed (e.g., ADD, AND, MOV, etc.).
- The Control Logic then generates microinstructions to activate the appropriate hardware units (buses, registers, ALU, etc.).
- If the instruction requires operands, the CU retrieves them from memory or internal registers by placing the corresponding address on the address bus, transferring the data to the Memory Register (MR).

For example, the instruction may request data located at address @0x101. The value (Data1) is fetched from memory and stored in the MR for use during execution.





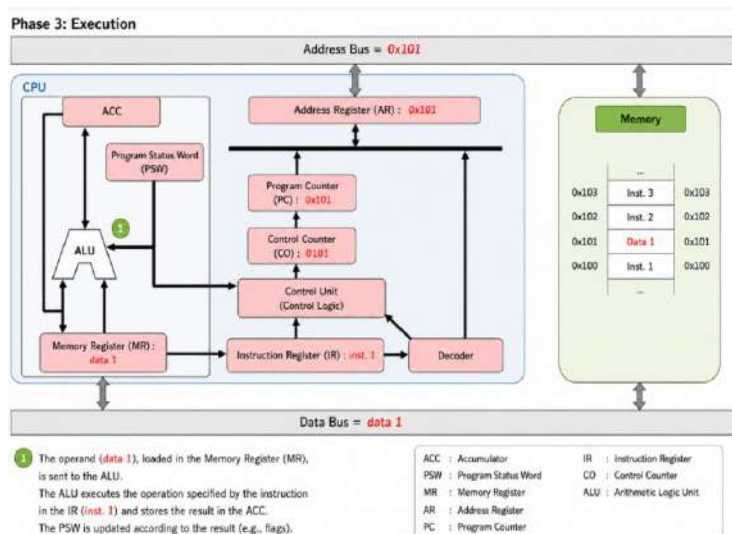


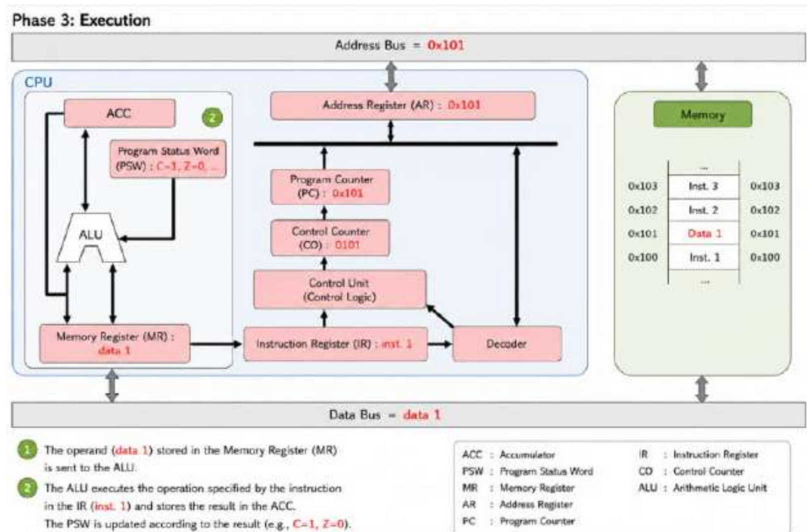
III.6.3 Execute Phase (Operation Execution)

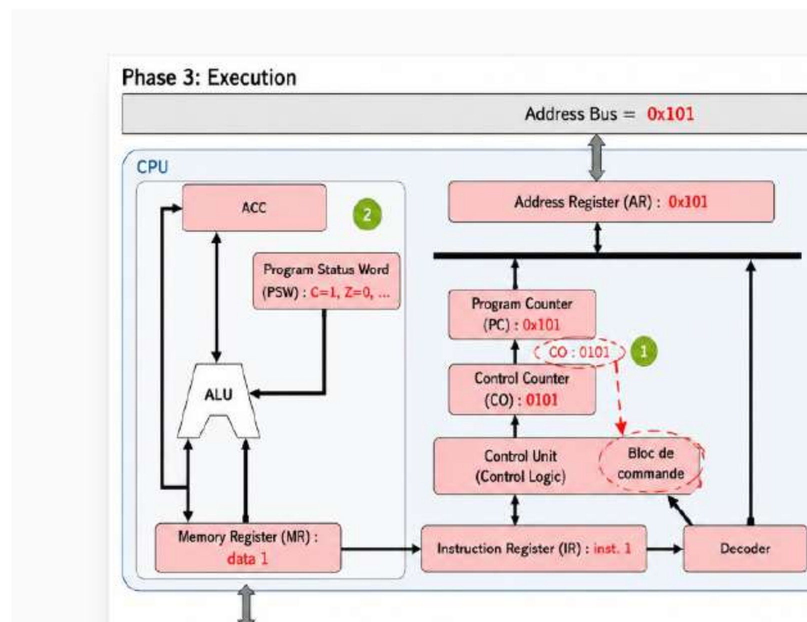
Once decoding and operand retrieval are complete, the execution phase begins:

- The operands are sent to the **ALU**, which performs the specified operation (arithmetic or logical).
- The result is stored in the **Accumulator (ACC)** or another destination register.
- The **Status Register (PSW)** is updated based on the result (e.g., Carry flag, Zero flag, Sign flag).
- If needed, the result can be sent back to **main memory** through the data bus.

In the corresponding diagram, the operand Data1 is processed by the ALU, which produces a result stored in the Accumulator. The Status Register is also updated accordingly (e.g., C=1, Z=0, etc.).







III.7 The ALU and Associated Registers

The lower part of Figure 75 shows the Arithmetic and Logic Unit (ALU) connected to registers A, B, and R. These registers play a central role in data processing:

- **Register A** holds the first operand,
- **Register B** holds the second operand,
- **Register R** stores the result after processing.

The logic gates NOT, AND, and OR—also visible in the figure—demonstrate that every complex operation relies on combinations of elementary logic functions. Thus, the ALU can

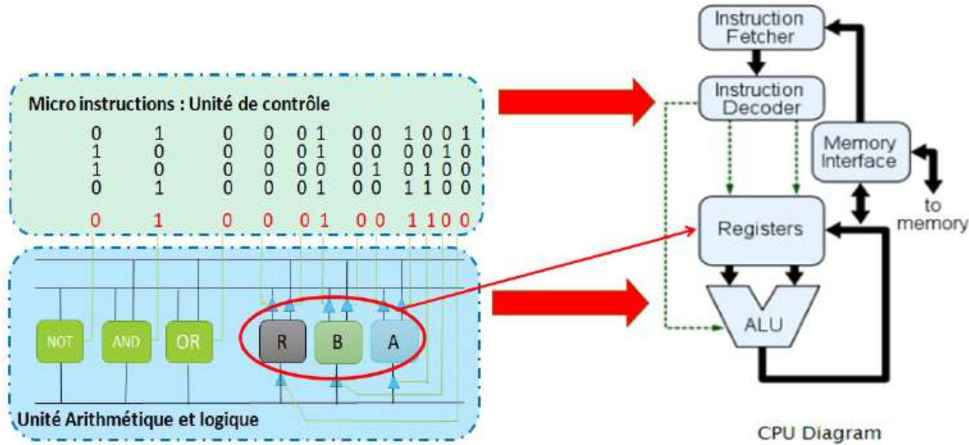
perform both logical and arithmetic computations by manipulating these basic building blocks.

III.7.1 General Processor Organization

The right part of Figure 75 presents a simplified CPU diagram comprising four main blocks:

- **Instruction Fetcher**: responsible for retrieving successive instructions from memory.
- **Instruction Decoder**: interprets the binary code of the instruction and translates it into a sequence of microinstructions.
- **Registers**: ensure temporary data storage and enable fast data exchange with the ALU.
- **ALU (Arithmetic Logic Unit)**: executes arithmetic and logical operations according to the microinstructions received.

These elements communicate through a memory interface, which enables data exchange between the processor and main memory.



III.7.2 CU–ALU Interaction

The red arrows in Figure 75 highlight the direct connection between microinstructions, registers, and the ALU. In practice:

- The Control Unit (CU) generates the microinstructions that activate the appropriate data paths and logic operations.

- The ALU executes the required operations on the values stored in A and B to produce a result in R.
- The CPU architecture integrates these results into the regular instruction execution flow.

Thus, the Control Unit acts as the conductor, ensuring step-by-step coordination, while the ALU performs the actual computations.

III.7.3 Processor Frequency and Clock Speed

The operation of a processor is governed by a periodic clock signal generated by an internal or external oscillator. This signal determines the rate at which all internal operations—fetch, decode, execute, and data transfers—are carried out.

➢ **Definition of Clock Frequency**

The clock frequency represents the number of clock cycles executed per second. It is expressed in hertz (Hz), but modern processors typically use higher multiples:

- MHz (Megahertz = 10^6 Hz)
- GHz (Gigahertz = 10^9 Hz)

Thus, a processor running at 3 GHz performs three billion clock cycles per second.

➢ **Clock Cycle and Instruction Execution**

Each clock cycle represents the smallest time unit in which the processor performs a basic operation (e.g., data transfer between registers, an elementary addition in the ALU, or the decoding of an instruction).

However, a complete instruction (such as an addition or a memory access) may require several clock cycles. For example:

- The fetch of an instruction typically takes one cycle,
- The decode stage another,
- The execution may take one or more cycles depending on operation complexity.

To measure efficiency, we define the CPI (Cycles Per Instruction) — the average number of clock cycles required to execute a single instruction.

➢ **Relationship Between Frequency and Performance**

Processor performance depends on multiple combined factors:

- Higher clock frequency → more operations per second.
- Lower CPI → improved efficiency through architectural optimizations such as pipelining, parallelism, and branch prediction.

Consequently, two processors running at the same frequency may exhibit very different performance levels if their internal architectures differ in optimization.

$$\text{Performance} \propto \frac{\text{Fréquence d'horloge}}{\text{CPI}}$$

- **Limits of Increasing Clock Frequency**

While increasing the clock frequency enhances execution speed, it introduces several constraints:

- Power consumption rises proportionally with frequency.
- Heat dissipation increases, requiring more efficient cooling systems.
- Signal stability decreases at very high speeds, as electrical signals may not propagate correctly through the circuitry.

These limitations led to a slowdown in the "frequency race" in the early 2000s and encouraged the adoption of multicore architectures, which increase computational power by multiplying processing units instead of endlessly raising clock speed.

- **Example: Evolution of Processor Frequencies**

- Early microprocessors (e.g., Intel 4004, 1971) operated at 740 kHz.
- Processors from the 1990s reached between 100 and 500 MHz.
- Modern desktop and server CPUs typically run between 2 GHz and 5 GHz, featuring multiple cores capable of working in parallel.

III.8 Types of Processors

III.8.1 Single-Core Processors

A single-core processor represents the simplest form of a processing unit. It contains only one computing core, meaning a single unit capable of fetching, decoding, and executing instructions. All operations—whether arithmetic calculations, memory management, or peripheral coordination—are performed sequentially by this single core.

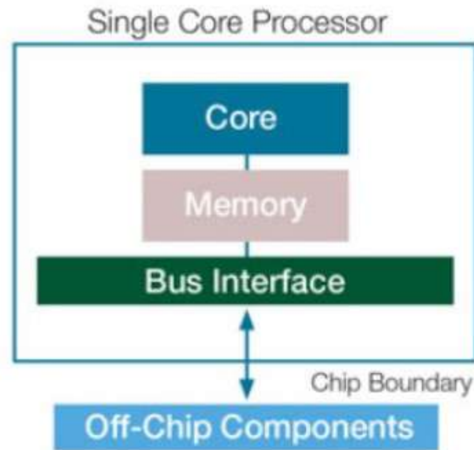


Figure III. 8 - Single Core Processor

Figure 1 illustrates this concept: the *Single Core Processor* is composed of one core connected to memory and a bus interface, which in turn communicates with external components (off-chip components). All instructions must therefore pass through this single core, which significantly limits performance when several applications run simultaneously.

Historically, single-core processors were sufficient for early personal computers and embedded systems, as applications were less complex and consumed fewer resources. However, with the growing demand for multimedia processing, scientific computing, and interactive applications, their execution speed limitations quickly became evident.

III.8.2 Multi-Core Processors (with Parallelism and Hyper-Threading)

The introduction of multi-core processors marked a major revolution in modern processor architecture. A multi-core processor integrates several computing cores within the same chip. Each core can independently execute its own instruction stream, allowing multiple tasks to be processed in parallel.

Figure 2 shows the architecture of a *Multi-Core Processor*: two cores (Core 1 and Core 2) can be seen, each with its local memory, while sharing a common memory and the same bus interface. This structure allows both independent execution and communication between cores.

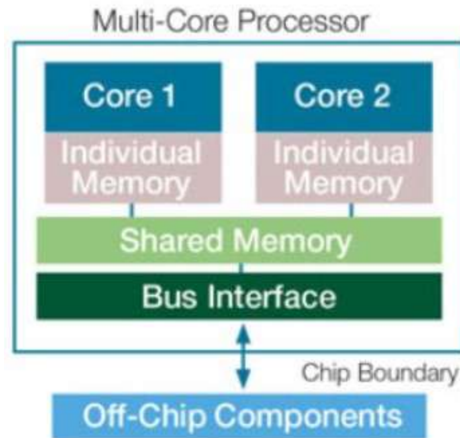


Figure III. 9 - Multi- core processor

Figure 3 illustrates the variety of available configurations: *Dual-core* (2 cores), *Quad-core* (4 cores), *Hexa-core* (6 cores), *Octa-core* (8 cores), and even *Multiple-core* designs (16, 32, or more). Such architectures are now found in almost all modern computers, servers, and smartphones.

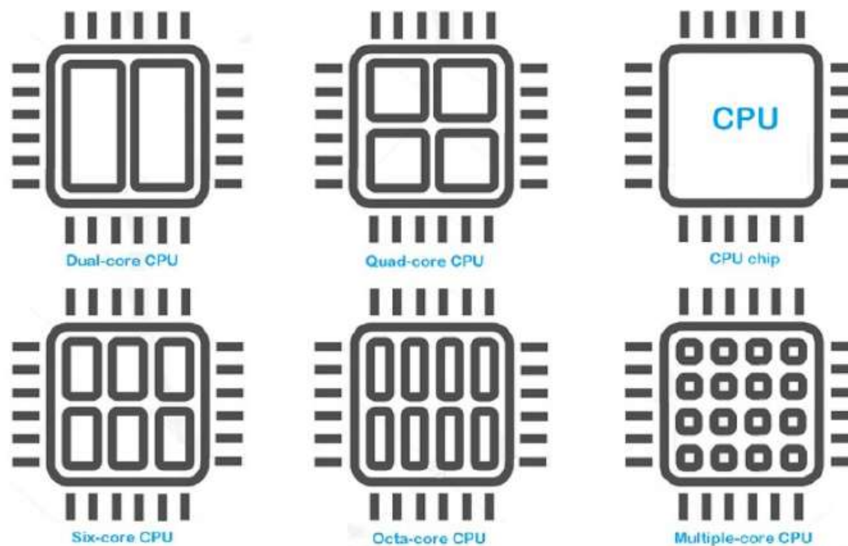


Figure III. 10 - hardware parallelism

The main advantage lies in hardware parallelism: several tasks can be executed simultaneously, significantly increasing the system's overall speed. For instance, one core can handle an office application while another processes a video stream. Moreover, with technologies such as Hyper-Threading (developed by Intel), each core can manage two virtual threads of execution, giving the impression that the processor has twice as many logical cores as physical ones.

However, overall performance also depends on the ability of software to exploit this parallelism. Poorly optimized applications may still use only a single core, thereby reducing the benefits of a multi-core processor.

III.9 CISC and RISC Processors

Beyond the number of cores, processors can also be classified according to their internal architecture. Two major philosophies dominate this domain: CISC and RISC.

CISC processors (Complex Instruction Set Computer) use a very large and complex set of instructions. Each instruction can perform sophisticated operations such as arithmetic processing and memory access within a single command. This simplifies programming, since a single assembly instruction can replace several simpler ones. The most well-known examples of CISC processors are those based on the x86 architecture (Intel, AMD). However, this complexity makes instruction decoding slower and can sometimes limit overall performance.

RISC processors (Reduced Instruction Set Computer) adopt an opposite approach. They rely on a small and optimized instruction set, where each instruction is designed to execute in a single clock cycle (or close to it). This simplicity allows for efficient pipelining, faster execution, and better optimization of parallel operations. Although more instructions are required to perform complex tasks, their speed of execution compensates for this apparent inefficiency. The ARM architecture, widely used in smartphones, tablets, and embedded systems, is a typical example of the RISC approach.

III.9.1 Instruction Sets

The instruction set represents the complete set of commands that a processor can execute. Each machine instruction is encoded in binary form and corresponds to a basic operation such as addition, data transfer, or comparison. The design of an instruction set directly influences the processor’s performance, power consumption, and hardware complexity. Two main architectural philosophies exist: CISC (Complex Instruction Set Computing) and RISC (Reduced Instruction Set Computing).

III.9.2 CISC (Complex Instruction Set Computing)

The CISC architecture is based on integrating into the processor a large number of complex instructions capable of performing in a single command what would otherwise require multiple simple instructions. A single instruction can therefore include several operations such as loading data from memory, performing arithmetic or logical processing, and storing the result back into memory.

Historically, the CISC approach was developed to make assembly language programming easier by bringing machine instructions closer to high-level programming languages. This allowed programs to be shorter—an important advantage when memory was limited and expensive.

Figure III. 11 - Instruction Execution Cycle in a CISC Architecture

As illustrated in Figure III.11, a CISC instruction goes through several steps: fetching the instruction from memory, decoding it, fetching one or more operands (Fetch D1, Fetch D2), executing the operation, and finally storing the result. These successive stages highlight the hardware complexity of CISC processors, which must decode and handle a very wide range of varied instructions.

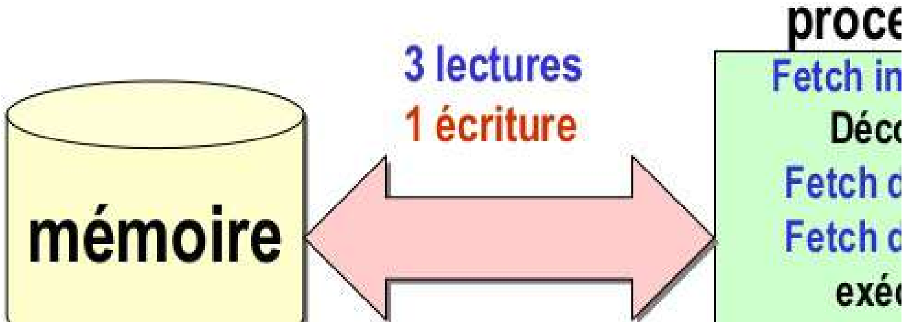


Figure III. 12 - Data and Instruction Flow Between Processor and Memory in a CISC Architecture

Moreover, Figure III.12 shows the relationship between the processor and memory: to execute a complex instruction, several read and write operations are required (for example, three reads and one write). Such frequent memory accesses explain why CISC architectures are often associated with longer access times, potentially slowing down execution despite their powerful instruction capabilities.

A concrete example is the instruction "ADD A, #2", illustrated in Figure 79. This single instruction performs three operations:

- Reading the immediate operand (#2);
- Adding this value to the content of register A;
- Storing the result directly back into A.

Thus, several stages are combined in one instruction, reducing the total program size in assembly. Figure III.13 also presents the binary format of this instruction, distinguishing the opcode field and the operand field.

Instruction (16 bits)	
Code opératoire (5 bits)	Champ opérande
ADD A	#2

Figure III. 13 - Example of a CISC Instruction Format (16-bit Instruction Encoding: Opcode and Operand Fields)

III.9.3 RISC (Reduced Instruction Set Computing)

In contrast, the RISC architecture is based on a reduced and highly optimized instruction set. Each instruction is designed to execute very quickly, typically within a single clock cycle. The principle is to simplify the hardware and delegate complexity to the software layer (compilers and programs).

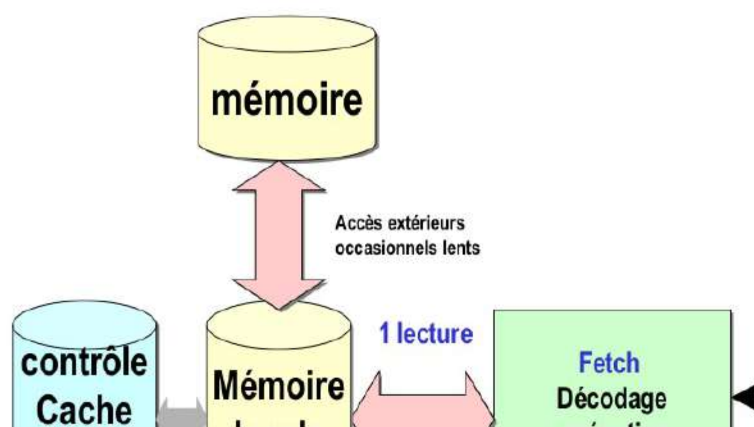


Figure III. 14 - Data Access and Execution Flow in a RISC Architecture (Using Local Memory and Register Operations)

As shown in Figure III.14 , data access occurs through fast internal registers, requiring only a single memory read instead of multiple accesses as in CISC systems. The execution

pipeline is much smoother, allowing efficient parallelization of stages (fetch, decode, execute). Instructions usually have a fixed length, which greatly simplifies the decoding process.

Thanks to this design philosophy, RISC processors achieve very high clock frequencies and excellent energy efficiency. This architecture is prevalent in most modern processors, especially those based on ARM, widely used in smartphones and embedded devices.

III.9.4 Comparison Between RISC and CISC

The difference between RISC and CISC goes beyond the number of instructions—it reflects two distinct design philosophies with different performance outcomes.

In CISC processors, each instruction is powerful and can perform several operations at once. However, this increases decoding complexity and the number of memory accesses (as shown in Figure III.11), which can slow down execution.

In RISC processors, instructions are simpler, shorter, and uniform. Execution relies primarily on internal registers (as shown in Figure III.13), reducing memory access and improving overall speed.

CISC emphasizes hardware complexity to reduce the number of software instructions, whereas RISC focuses on hardware simplicity and relies on software optimization to make efficient use of the processor.

Modern architectures tend to combine the advantages of both approaches:

- **x86 processors** (originally CISC) internally translate complex instructions into **micro-operations** closer to RISC style.
- **ARM processors** (RISC) have gradually integrated more complex instructions to enhance performance for specific applications.

III.10 Optimizations in Modern Processors

The evolution of processors relies on several optimization techniques aimed at increasing instruction throughput and reducing waiting times, without merely raising the clock frequency.
These optimizations include pipelining, superscalar and parallel execution, the use of a cache memory hierarchy, as well as speculative execution and branch prediction.

III.10.1 Pipeline (Superpipeline, Out-of-Order Execution)

The pipeline is an optimization technique that divides the execution of an instruction into several successive stages: fetch, decode, execute, memory access, and write-back.

As illustrated in Figure III. 15 , this approach allows the processor to begin a new instruction before the previous one has finished executing. Thus, several instructions are processed simultaneously at different stages of the pipeline, greatly increasing the overall throughput.

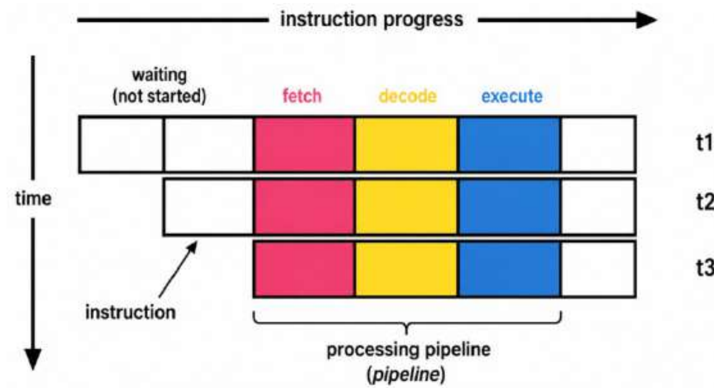


Figure III. 16 - Principle of Pipeline Operation

Figure III. 16 compares the execution time with and without pipelining. Without a pipeline, execution is strictly sequential, and total time is proportional to the number of instructions. With a pipeline, stages overlap, significantly reducing the overall execution time.

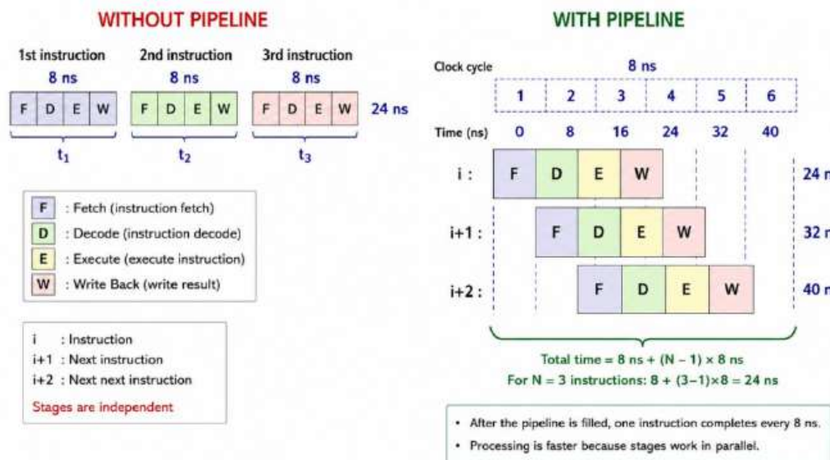


Figure III. 17 - Execution Time with and without Pipeline

However, pipeline efficiency is limited by several constraints:

Each stage must have a similar duration to prevent bottlenecks.

- All instructions must pass through the same stages, even if some are simpler.
- Hazards may occur: data dependencies, memory access conflicts, or control hazards (branching).

To improve performance, two major enhancements have been introduced:

- Superpipeline: the pipeline is divided into a greater number of stages (14, 20, or more), which increases the clock frequency and throughput (e.g., Intel Pentium 4 with 20 stages).
- Out-of-Order Execution: independent instructions can be executed ahead of others waiting for resources (e.g., memory access), maximizing the utilization of functional units.

Figure III. 17 illustrates the five classic stages of a RISC pipeline (IF, ID, EX, MEM, WB), which served as the foundation for modern architectures.

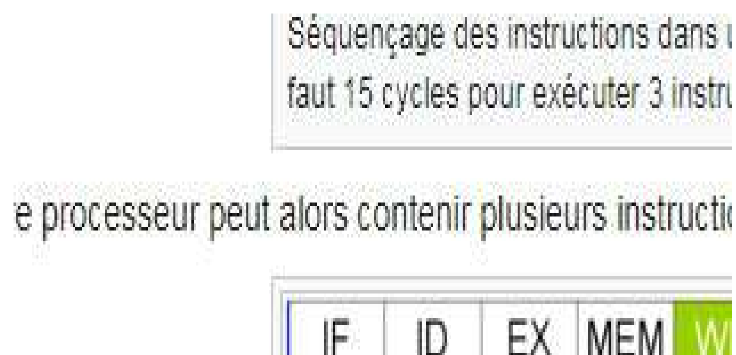


Figure III. 18 - Stages of a Classical RISC Pipeline

Figure III. 18 shows the sequencing of multiple instructions in a five-stage pipeline. Without pipelining, 15 cycles are required to execute three instructions, compared to only 7 cycles with pipelining.

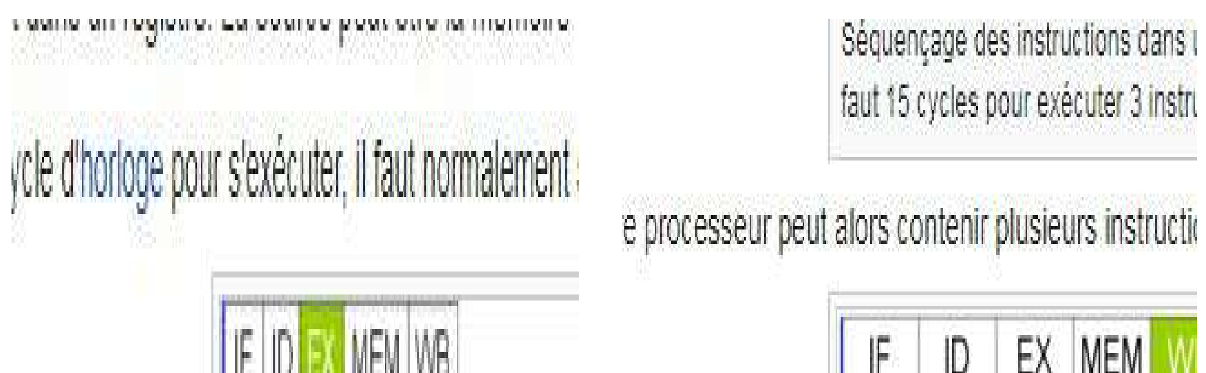


Figure III. 19 - Instruction Sequencing with and without Pipeline

Examples of pipeline depth in different generations of processors:

- Intel Pentium 4: 20 stages
- Intel Pentium II: 14 stages
- AMD Athlon: 12 stages
- Motorola PowerPC G4: 7 stages

III.10.2 Superscalar Architecture and Parallel Execution

While pipelining improves execution speed, it is still limited to processing one instruction per cycle. The superscalar architecture goes further by allowing multiple instructions to be executed simultaneously per cycle through several functional units (ALU, FPU, memory units, etc.).

Examples:

- A dual-issue processor executes two instructions per cycle.
- A quad-issue processor executes four.

Instructions are automatically distributed among the available units depending on their dependencies and type. An extension of this concept is Simultaneous Multithreading (SMT), best known through Intel's Hyper-Threading technology. This technique presents one physical core as two logical cores, improving resource utilization and apparent parallelism.

III.10.3 Cache Memory (L1, L2, L3) and Memory Hierarchy

Processor speed today far exceeds that of main memory (RAM), creating a major bottleneck in performance. To reduce this gap, modern processors use a hierarchical cache memory system (Figure III.19):

- L1 Cache: integrated into each core, extremely fast but small in size (32–128 KB). Usually divided into instruction and data caches.
- L2 Cache: larger (512 KB to several MB), slightly slower, either dedicated to one core or shared between two.
- L3 Cache: much larger (up to several tens of MB), shared among all processor cores.

The processor checks L1 first, then L2, then L3, and finally the main memory (RAM). This hierarchical access significantly reduces the average data access time for instructions and operands.

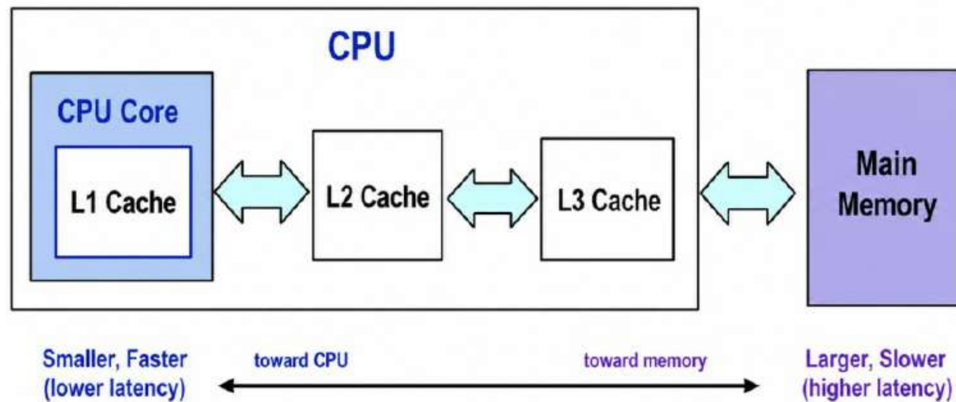


Figure III. 20 - Cache Memory Hierarchy (L1, L2, L3) and Relation with RAM

This system exploits two key principles:

- Temporal locality: reuse of recently accessed data.
- Spatial locality: access to memory locations that are physically close to one another.

III.10.4 Speculation and Branch Prediction

Conditional branch instructions (if, loop, jump) represent a significant challenge, as the processor must determine in advance which execution path to take. Waiting for the branch condition to resolve would flush the pipeline and slow down performance.

To avoid this, modern processors employ:

- Branch Prediction: the processor predicts the likely outcome of a branch based on past execution history.
- Speculative Execution: the processor executes instructions along the predicted path before the actual branch condition is confirmed.

If the prediction is correct, significant time is saved; if not, the speculative results are discarded (pipeline flush) and the correct path is executed. Thanks to sophisticated dynamic prediction algorithms, modern Intel and AMD processors achieve over 95% prediction accuracy, ensuring a nearly continuous instruction flow.

III.10.5 RISC-Based Boards

RISC processors, due to their simplicity and energy efficiency, are widely used in embedded systems, smartphones, tablets, and even modern servers.

- ARM (Advanced RISC Machine): currently the most widely used RISC architecture in the world. Boards such as the Raspberry Pi and STM32 microcontrollers are based on ARM cores. They offer an excellent balance between low power consumption and computing performance.
- RISC-V: an open and royalty-free RISC architecture, increasingly used in education, research, and industrial projects. Boards such as the HiFive Unmatched or the BeagleV are based on this architecture.
- Typical applications: Internet of Things (IoT), real-time systems, robotics, embedded data processing, and edge computing.

III.11 How to Choose and Integrate a RISC/CISC Board into a Project

The choice between a RISC or CISC-based board depends on several factors related to the project context, technical constraints, and final objectives.

Criterion	RISC (ARM, RISC-V, etc.)	CISC (x86, Intel, AMD, etc.)
Power consumption	Low consumption, ideal for embedded systems, IoT devices, and battery-powered applications.	Higher power consumption, suitable for workstations, PCs, and servers requiring high performance.
Software compatibility	Growing support (Android, Linux ARM, RISC-V, modern compilers) but sometimes requires adaptation.	Large, mature software ecosystem (Windows, Linux, professional applications).
Computing power and parallelism	Optimized for specific computations, IoT, robotics, distributed and real-time systems.	Very high performance for intensive computing, graphics rendering, simulation, databases, and virtual machines.
Cost and accessibility	Low cost and widely available (Raspberry Pi, STM32, RISC-V boards), ideal for prototyping and education.	More expensive, but provides higher performance and universal compatibility.

Integrating a board into a project requires:

- Evaluating hardware resource requirements: memory, storage, and interfaces (GPIO, USB, Ethernet, etc.).

- Choosing the appropriate operating system: embedded Linux (e.g., Yocto, Ubuntu ARM) for RISC boards, or standard Windows/Linux for CISC boards.

Optimizing the software architecture and code:

- Use libraries specific to ARM or RISC-V architectures.
- Leverage **SSE/AVX** optimizations on x86 processors.

Considering scalability: a large-scale IoT project may use low-cost RISC-V cores, while a high-performance computing project may require an x86 platform.

III.12 Processors According to Application Domains

Modern processors are no longer limited to personal computers. They now exist in multiple variants tailored to various environments, such as PCs, smartphones, embedded systems, IoT devices, and high-performance computing (HPC) platforms.

III.12.1 Microprocessors for Computers

Desktop and laptop computers are mainly based on the x86 architecture, dominated by Intel and AMD. These processors are characterized by:

- High computing power with a wide range of frequencies and cores.
- Universal software compatibility (Windows, Linux, macOS via translation, professional software).
- Advanced features: hyper-threading, power management, and large cache memory.

Examples: Intel Core i3/i5/i7/i9, AMD Ryzen 5/7/9. These architectures are suited for demanding applications such as advanced office work, gaming, CAD, and scientific simulation.

III.12.2 Microprocessors for Smartphones

Smartphone processors are primarily based on the ARM architecture, known for its low power consumption.

- Qualcomm Snapdragon, Samsung Exynos, and Apple A/M series are the main examples.
- They integrate multiple types of cores (high-performance + low-power) in big.LITTLE architectures to optimize battery life.

- These processors also embed GPU and AI accelerators for facial recognition, computational photography, and augmented reality.

Example: The Apple M1/M2, based on ARM, now rivals x86 processors in laptops.

III.12.3 Microprocessors for Embedded Systems and IoT

In embedded and IoT systems, power efficiency and low cost are more important than raw computing power.

- ARM Cortex-M: a family of processors specialized in real-time control, used in STM32, Arduino DUE, and similar boards.
- RISC-V: an open-source architecture gaining ground thanks to its flexibility and license-free model, fostering academic and industrial innovation.
- ESP32 by Espressif: an example of an IoT processor integrating Wi-Fi and Bluetooth, widely used in connected projects.

These processors power millions of connected devices: watches, sensors, drones, and medical equipment.

III.12.4 Microprocessors for High-Performance Computing and Artificial Intelligence

To meet the massive computational demands of scientific computing and artificial intelligence, specialized processors have been developed:

- GPU (Graphics Processing Unit): originally designed for graphics rendering, now used for deep learning and massive parallel computation. Example: NVIDIA CUDA.
- TPU (Tensor Processing Unit): developed by Google, optimized for neural network operations (matrix and tensor computations).
- NPU (Neural Processing Unit): integrated into some smartphones and servers to accelerate AI tasks such as voice or image recognition.

These specialized processors play a crucial role in High Performance Computing (HPC), data centers, and modern AI applications.

III.13 Comparison of Modern Processor Architectures

The recent evolution of processors has led to a wide diversity of architectures, each suited to specific needs and contexts.

The following table summarizes the main characteristics of modern processor families:

Architecture	Examples	Strengths	Limitations	Application Domains
x86 (Intel, AMD)	Intel Core i9, AMD Ryzen 9	High computing power, universal software compatibility, advanced multitasking (SMT, Hyper-Threading)	High power consumption, costly, generates heat	Desktop PCs, workstations, high-performance servers
ARM (Cortex-A, Cortex-M, Apple M1/M2, Snapdragon)	Apple M1/M2, Qualcomm Snapdragon, STM32	Low power, high energy efficiency, integrated GPU/AI, widely used in embedded systems	Less powerful for heavy computation (except Apple Silicon), limited compatibility in some cases	Smartphones, tablets, IoT, low-power servers
RISC-V (open source)	SiFive, Kendryte K210	Open and modular architecture, no license, customizable design, strong academic adoption	Developing software ecosystem, limited high-end performance	Microcontrollers, embedded systems, R&D, IoT
GPU (Graphics Processing Unit)	NVIDIA CUDA, AMD Radeon	Massive parallelism, ideal for AI and graphics, high performance for deep learning	High power usage, memory dependency, not suited for sequential tasks	Graphics, AI, HPC, scientific simulation
TPU (Tensor Processing Unit)	Google TPU	Optimized for neural networks, efficient for matrix and AI computations	Specialized, mainly cloud-based, limited public access	Artificial intelligence, machine learning, data centers
NPU (Neural Processing Unit)	Huawei Kirin NPU, Qualcomm Hexagon	On-chip AI acceleration, low power for specific tasks	Lower performance than GPU/TPU, targeted use cases	Smartphones, IoT devices, edge AI processing
Quantum Processors	IBM Q, Google Sycamore	Exponential computational capability for specific problems (cryptography, simulation)	Experimental technology, qubit instability, limited usage	Scientific research, cryptography, molecular simulation
Neuromorphic Processors	Intel Loihi	Brain-inspired design, excellent energy efficiency for cognitive AI	Emerging technology, not yet widely deployed	AI research, robotics, intelligent systems