Higher School of Applied Sciences – Tlemcen

Department of Second Cycle Education

Program: *Industrial Engineering*

Course Handout

Embedded Electronics – Course Handout

Instructor: Dr. Hicham MEGNAFI

Academic Year: 2025 / 2026

Preface

This course handbook is intended for first-year graduate students specializing in Industrial Engineering.

Its objective is to introduce the fundamental principles of embedded electronics, with a focus on understanding the operation of microcontrollers, hardware architectures, and communication interfaces used in modern embedded systems.

Through this module, students will explore how embedded systems form the core of intelligent and automated devices employed in industrial environments. They will learn to analyze the structure of an embedded system, understand the role of hardware components (processor, memory, buses, input/output interfaces), and grasp the basics of their integration within an industrial context.

This course aims to develop a comprehensive perspective that connects electronics, programming, and automation, preparing future engineers to design and operate modern, connected industrial systems.

Table of Contents

Chapter I: Introduction to Embedded Electronics

I Introduction

I.1 Context of Embedded Electronics

Embedded electronics, a cornerstone of modern systems, addresses major industrial challenges by integrating three key components: sensors, microcontrollers or processors, and actuators.

Challenge: In industrial environments, the main objectives are to optimize operations, reduce maintenance costs, and improve productivity. Embedded electronic systems play a crucial role in achieving these goals by enabling intelligent control, real-time monitoring, and autonomous decision-making across machines and processes.

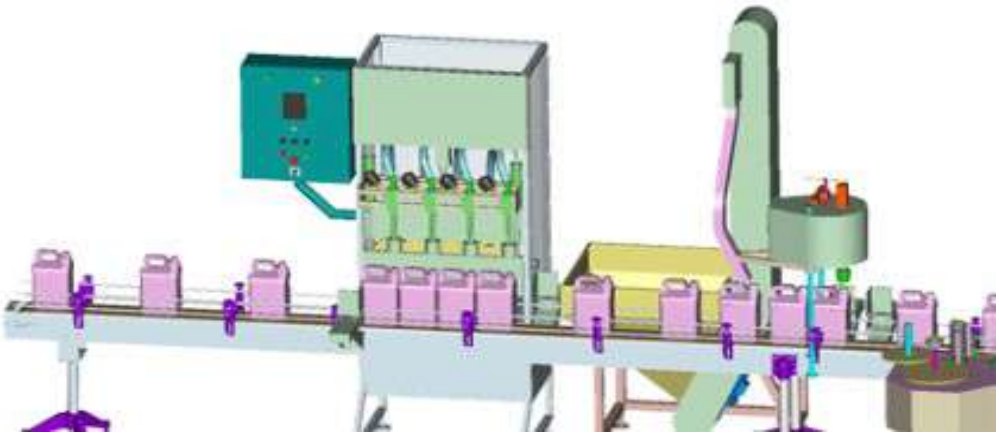


Figure I. 1 - Example of an Automated Production Line Using Embedded Electronics

Proposed Solutions:

I.1.1 Solution Based on Combinational Logic:

Advantages: Conceptual simplicity and low initial cost.

Limitations: Lack of flexibility for complex tasks and difficulty in adapting to system modifications or evolving functional requirements.

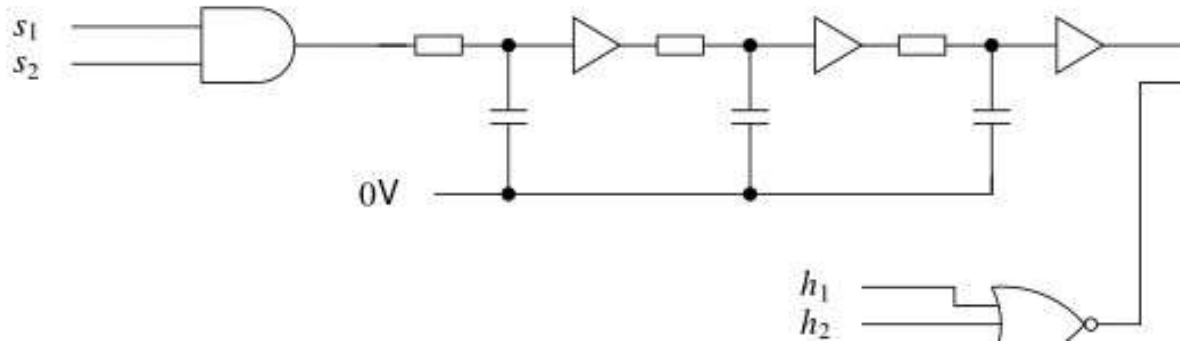


Figure I. 2 - Combinational Logic Diagram

I.1.2 Solution Based on Traditional Electronics:

Advantages: High adaptability to changing requirements and efficient management of complex systems.

Limitations: Increased costs due to circuit complexity, the need for advanced technical expertise, and hardware constraints that may limit scalability or integration.

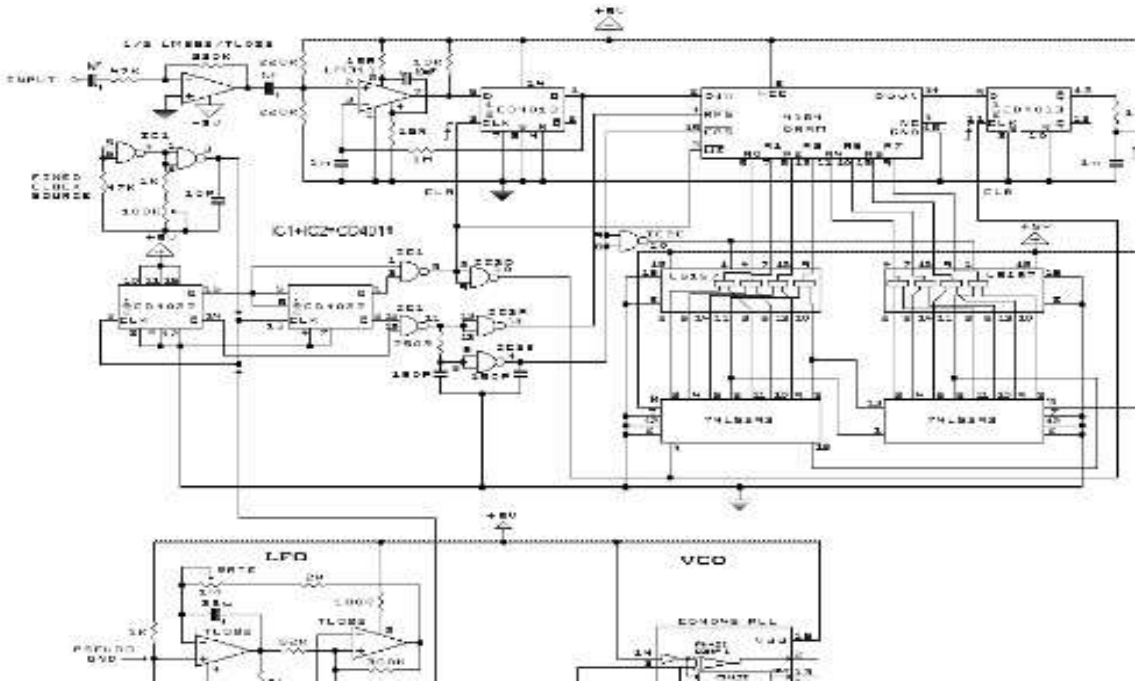


Figure I. 3 - Traditional Electronics Model

I.1.3 Solution Based on Embedded Systems:

Advantages: High flexibility, capability to manage complex tasks, and connectivity enabling remote monitoring and control.

Limitations: Higher initial cost and reliance on specialized technical expertise.

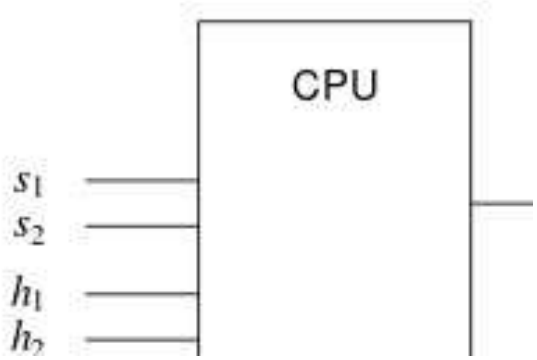


Figure I. 4 - Architecture of an Embedded System

Overcoming Limitations through the Transition from Hardware to Software (Embedded Systems):

- **Initial Costs:** Although the upfront investment in embedded systems may be higher, it is offset by reduced maintenance and operational costs in the long term.
- **Complexity:** Embedded systems offer programmability that allows for more flexible and scalable task management, reducing operational complexity.
- **Dependence on Expertise:** Thanks to user-friendly programming interfaces and advanced development environments, embedded systems have become more accessible, lowering the need for deep hardware expertise.
- **Flexibility:** The programmability of embedded systems enables rapid adaptation to industrial process changes, providing superior flexibility compared to fixed hardware solutions.
- **Connectivity and Remote Monitoring:** Embedded systems support networking capabilities, enabling remote supervision, centralized control, and seamless integration with other digital systems.

The shift toward embedded systems thus represents a transition to a more intelligent, adaptable, and software-oriented approach, where programmability and connectivity effectively overcome the traditional limitations of purely hardware-based designs.

I.2 Projects Developed at Our School

I.2.1 Project 1: Automated Attendance Management System

This project demonstrates a practical application of RFID (Radio Frequency Identification) technology in the educational domain. It involves the design and implementation of an automated attendance management system aimed at accurately monitoring student attendance within academic institutions. The main objective is to facilitate the identification of absentee students and provide reliable indicators of attendance for teachers and administrators.

The system architecture is composed of three integrated components:

1. **Electronic Component:** Each student carries an RFID card. Upon entering the classroom, the card is detected by an RFID reader connected to an Arduino UNO microcontroller. The reader captures the radio frequency signal emitted by the card and sends the student's ID to the Arduino for preliminary processing, such as activation and attendance recording.
2. **Telecommunication Component**: The attendance data collected by the Arduino are transmitted to a central server via an Ethernet module. This network link acts as the communication bridge between the physical hardware and the information system.
3. **Software Component**: The software part includes a server, a database, and a web application. The database stores student, teacher, course, and schedule data. The web interface—developed using PHP/MySQL and HTML/CSS—enables real-time attendance tracking, report generation, and statistical analysis. It provides multi-platform access from computers, tablets, and smartphones.

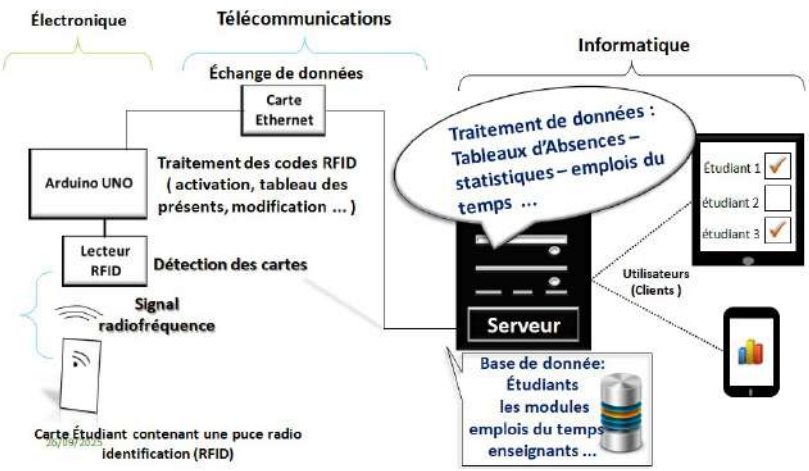


Figure I. 5 - General architecture of the automated attendance management system based on RFID technology.

This figure illustrates how the RFID card is detected by the reader connected to the Arduino, the data are then transmitted through the Ethernet module to the server, which processes and stores the information in the database. The web application allows teachers and administrators to visualize attendance in real time and generate statistical reports. The diagram highlights the synergy between electronics, telecommunications, and computer science in developing a reliable embedded system.

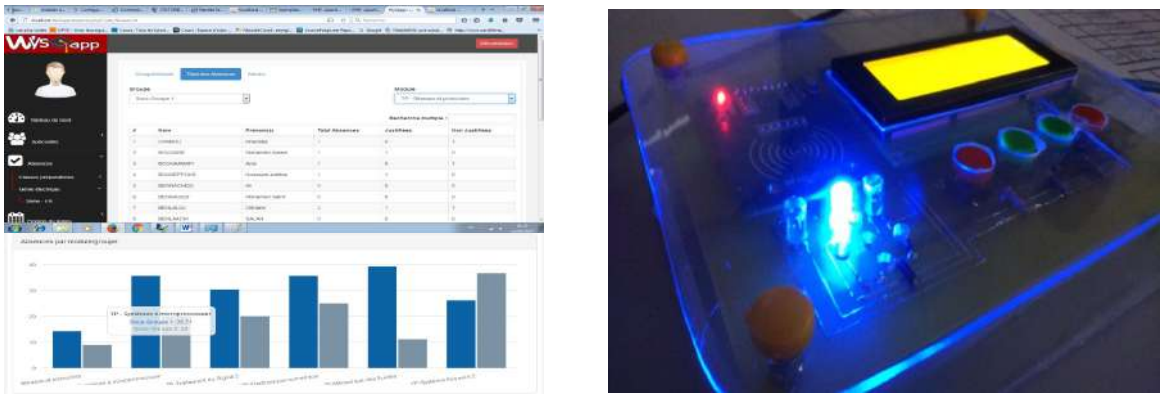


Figure I. 6 - Developed hardware prototype and associated software interface for attendance management and monitoring.

The prototype demonstrates the physical implementation using Arduino and the RFID reader, while the web interface showcases the available functionalities for attendance tracking, schedule visualization, and statistical reporting. Together, they validate the feasibility and practical relevance of the proposed system in real-world educational settings.

This work was presented under the title: “Design and Implementation of an Embedded System for Effective Attendance Management”, at the NewMat’21 – 1st International Conference: New Trends on Innovative Construction Materials, held at ESSA Tlemcen (Algeria), on March 22–23, 2022.

I.2.2 Project 2: Development of a 3D Printer – Version 1

The second project conducted as part of the embedded systems course focused on the design and implementation of a first-version 3D printer. The idea emerged from the need to produce custom plastic components for scientific projects while significantly reducing the high costs associated with commercial 3D printing services. As engineering students, the

proposed solution was to build the printer from scratch using readily available components and open-source software.

The main objective was to develop a fully functional 3D printer capable of manufacturing plastic prototypes using materials such as PLA and ABS. The project provided an opportunity to explore multiple embedded system concepts, particularly:

- **Motor control**: precise positioning through stepper motors for the X, Y, and Z axes.
- **Thermal regulation:** control of the heating extruder and heated bed.
- **System interfacing**: communication between the microcontroller and the user interface for process monitoring and control.

At the core of the system lies an Arduino Mega microcontroller equipped with a RAMPS 1.4 shield and stepper motor drivers to manage movement. Limit switches were integrated to calibrate the axes and ensure accurate positioning.

On the software side, the Marlin firmware was installed on the Arduino to interpret G-code commands generated by slicing software such as Cura or Slic3r. This allows the user to go seamlessly from a 3D model designed on a computer to a physical printed object, with real-time monitoring and control of printing parameters through a PC interface.

The prototype, shown in Figure 7, includes the mechanical structure, the electronic control board, and the power modules. This initial version successfully achieved the printing of small test objects, confirming the technical feasibility of the system. The project also demonstrated the pedagogical value of 3D printing, as it enables rapid prototyping—turning ideas into tangible models—while reducing manpower and tooling requirements.

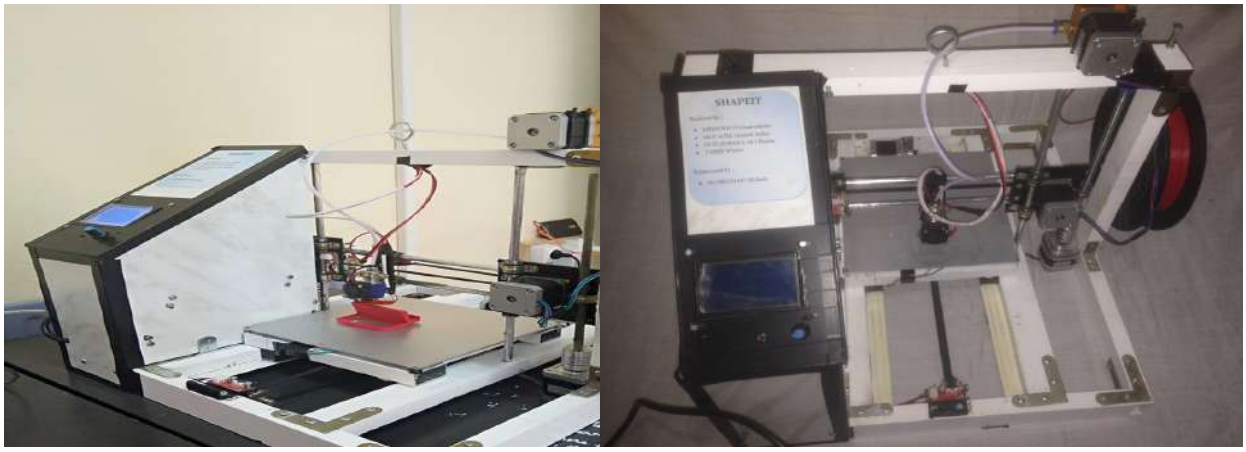


Figure I. 7 - Prototype of the 3D printer (Version 1) developed by engineering students, showing the mechanical frame, control electronics, and software interface.

This first version represents a foundational step in the development of a 3D printer. Future improvements could focus on increasing print speed, enhancing print quality, and improving overall reliability. Beyond its technical aspects, this project exemplifies the integration of mechanical, electronic, and software components within a complex embedded system. It also highlights the interdisciplinary nature of embedded engineering, combining control theory, automation, and digital manufacturing.

This work was presented in the following publication: H. Megnafi, O. Ayad, W. Tabib, A. A. Mouaziz, R. Ould Babaali, I. Medjhoud, "Improved Printing Time by Changing the Mechanical Part of the 3D Printer, Embedded System Application," NewMat'21 – 1st International Conference: New Trends on Innovative Construction Materials, ESSA-Tlemcen (Algeria), March 22–23, 2022.

I.2.3 Project 3: Development of a 3D Printer – Version 2

The second version of the 3D printer was designed with the goal of improving the performance achieved with the initial prototype. This enhanced version integrates a more efficient mechanical architecture, resulting in a threefold increase in printing speed and greater precision in the fabrication of printed parts.

The control system is built around an Arduino Mega 2560 microcontroller combined with the RAMPS 1.4 shield, which serves as the interface between the controller and the actuators. The printer's movement along the X, Y, and Z axes is handled by NEMA 17 stepper motors, driven by A4988 drivers that regulate the current and ensure precise motion control.

On the mechanical side, the structure was redesigned to minimize vibration and improve rigidity. A heated bed was added to enhance the adhesion of printed parts during the initial layers, while the hotend nozzles reach temperatures suitable for a variety of materials such as PLA and ABS. Limit switches were also integrated to establish reference positions for each axis, allowing for automatic calibration.

From an interface and autonomy perspective, a LCD screen with an SD card reader was implemented, enabling standalone operation without the need for a computer connection during printing. A 12V – 20A power supply provides sufficient energy for the heated bed, motors, and extruder heaters. The system operates under the Marlin firmware, configured to support dual-extruder management and optimized axis movement. The G-code files, generated through slicing software such as Cura or Repetier-Host, are directly interpreted by the Arduino, which controls the full printing sequence autonomously.

These hardware and software upgrades significantly reduced printing time while enhancing surface quality, consistency, and precision. This project demonstrates how refining both mechanical and electronic architectures can dramatically improve the performance of a complex embedded system such as a 3D printer.

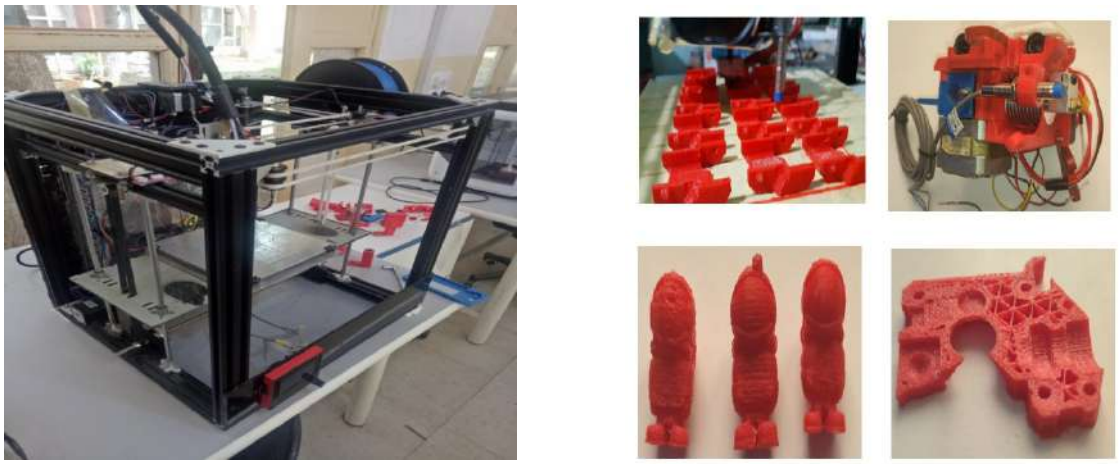


Figure I. 8 - 3D Printer – Version 2.

This work was presented in the following publication: H. Megnafi, O. Ayad, W. Tabib, A. A. Mouaziz, R. Ould Babaali, I. Medjhoud, "Improved Printing Time by Changing the Mechanical Part of the 3D Printer, Embedded System Application," NewMat'21 – 1st International Conference: New Trends on Innovative Construction Materials, ESSA-Tlemcen (Algeria), March 22–23, 2022.

I.2.4 Project 4: Design and Implementation of an Autonomous Watering System Based on PIC18F452

This project focuses on the design and realization of an autonomous irrigation system aimed at optimizing water management for agricultural and gardening applications. The system is entirely self-powered, operating on solar energy through a solar panel coupled with a rechargeable battery, thus ensuring complete energy autonomy. The PIC18F452 microcontroller serves as the central processing unit, managing data acquisition, decision-making, and the control of various sensors and actuators.

The system integrates soil moisture sensors and an LM35 temperature sensor to continuously monitor environmental and soil conditions. Based on predefined thresholds, the microcontroller automatically controls electrovalves to activate watering when necessary. This automated process ensures that plants receive the right amount of water while preventing waste, contributing to more efficient and sustainable irrigation management.

The user interface includes a 16×2 LCD display and push buttons, allowing users to select operation modes (automatic or manual) and configure system parameters. Furthermore, a Wi-Fi communication module enables remote supervision and control, allowing the user to monitor environmental data and system status via a smartphone or computer.

Figure I. 9 - Block diagram of the autonomous watering system.

The block diagram illustrates the main components of the system: the solar energy unit (panel and battery) supplies power to the control circuit; the sensors provide environmental data to the PIC18F452, which processes the inputs and drives the electrovalves accordingly. The LCD display and Wi-Fi module facilitate local and remote interaction, respectively.

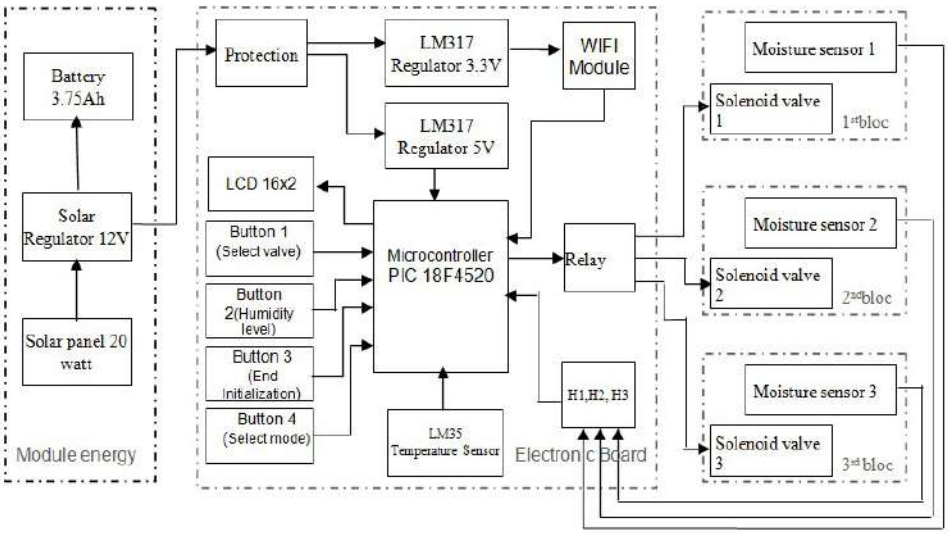


Figure I. 10 - Prototype of the autonomous watering system.

The developed prototype demonstrates the integration of hardware and software subsystems in a compact, functional design. It validates the feasibility of an energy-autonomous, intelligent irrigation controller capable of real-time adaptation to environmental changes.

This project exemplifies how embedded systems can be leveraged to address sustainability challenges by combining renewable energy, automation, and intelligent control.

This work was presented in the following publications:

- A. Chellal, H. Megnafi, A. Benhanifia, Design and Conception of an Autonomous Watering System, Garden (a Multi-Application Watering System), NewMat'21 – 1st International Conference: New Trends on Innovative Construction Materials, ESSA-Tlemcen (Algeria), March 22–23, 2022.
- H. Megnafi, A. A. Chellal, A. Benhanifia, *Flexible and Automated Watering System Using Solar Energy*, in *Artificial Intelligence and Renewables Towards an Energy Transition 4*, Springer International Publishing, 2021, pp. 747–755.

I.2.5 Project 5: Development of an Intelligent Urban Lighting System

This project focuses on the design and implementation of an IoT-based smart public lighting system aimed at significantly reducing energy consumption while improving operational efficiency and sustainability in urban environments. The developed prototype utilizes an ESP8266 NodeMCU microcontroller, which manages and coordinates the operation of the various modules within the system.

The system's power supply is provided by a photovoltaic solar panel, connected to a voltage regulator and a charging circuit that stores energy in a lithium battery, ensuring complete energy autonomy. This solar-based design not only supports renewable energy use but also enables continuous operation even in areas without access to the electrical grid.

A PIR (Passive Infrared) motion sensor is integrated for presence detection, allowing the system to automatically adjust lighting intensity based on pedestrian or vehicle movement. When no movement is detected, the light operates at a reduced intensity to conserve energy; it brightens immediately upon motion detection. A 16×2 LCD display provides real-time system status and data visualization, while a high-efficiency LED module is used for illumination.

The ESP8266 microcontroller, equipped with built-in Wi-Fi capability, ensures IoT connectivity and remote supervision. This allows monitoring and control through an online dashboard, enabling urban management authorities to track energy consumption, system performance, and maintenance needs in real time.

This smart lighting architecture integrates renewable energy, intelligent control, and wireless communication, resulting in a system that is energy-efficient, autonomous, and well-suited for smart city applications.

Figure I. 11 - Simplified schematic of the intelligent urban lighting system and the developed prototype.

The figure illustrates the system's functional structure, showing the interaction between the solar power subsystem, the control and sensing units (ESP8266 and PIR sensor), and the LED lighting module. It also depicts how the IoT layer enables remote communication and system monitoring.

This work was presented in the following publication: Imen Souhila Bousmaha, Hicham Megnafi, Hamza Benhadouga, Imene Hemarid, IoT Applications in Smart Public Lighting Management, The International Conference on Applied Science and Engineering (ICASE-22).

I.3 Fundamentals of Microcontrollers and Microprocessors

I.3.1 Role and Importance of Microprocessors

The microprocessor is the central processing unit (CPU) of a computing system, designed to execute general-purpose tasks. It operates based on either a Von Neumann or Harvard architecture and is composed mainly of three essential elements: the Arithmetic and Logic Unit (ALU), the Control Unit (CU), and a set of internal registers. These components work together to process instructions stored in memory, enabling the execution of complex computational sequences.

The microprocessor functions by fetching, decoding, and executing instructions, managing multiple tasks efficiently through the coordination of its control unit and the support of a multitasking operating system. Modern microprocessors integrate advanced architectural concepts such as multi-core structures, instruction pipelining, and hierarchical cache memory systems (L1, L2, L3) to optimize performance and reduce latency. Many also include specialized processing units, such as SIMD (Single Instruction, Multiple Data) and integrated GPUs, to accelerate parallel data processing and graphical computations.

The importance of microprocessors lies in their high computational capability and flexibility. They constitute the core of a board spectrum of applications, ranging from personal computing systems and servers to large-scale data centers and embedded systems that demand substantial processing capabilities. In the context of embedded electronics, microprocessors are employed in advanced systems such as autonomous vehicles, robotics, industrial control systems, and smart IoT devices, where they handle real-time signal processing, machine learning algorithms, and complex control operations.

In summary, the microprocessor represents the computational intelligence of modern digital and embedded systems, combining speed, precision, and adaptability to meet the demands of high-performance applications.

I.3.2 Role and Importance of Microcontrollers

A microcontroller is an integrated solution specifically designed to perform control and monitoring functions within embedded systems. Unlike general-purpose processors, it brings together on a single chip a CPU, various types of memory (ROM, RAM, EEPROM or Flash), and a set of input/output peripherals that allow direct interaction with the physical environment.

In addition to its core processing unit, a typical microcontroller includes functional modules such as analog-to-digital converters (ADC), digital-to-analog converters (DAC), timers, and serial communication interfaces (UART, SPI, I²C, CAN). Some modern versions even embed wireless communication modules like Bluetooth or Wi-Fi, enabling remote control and IoT integration.

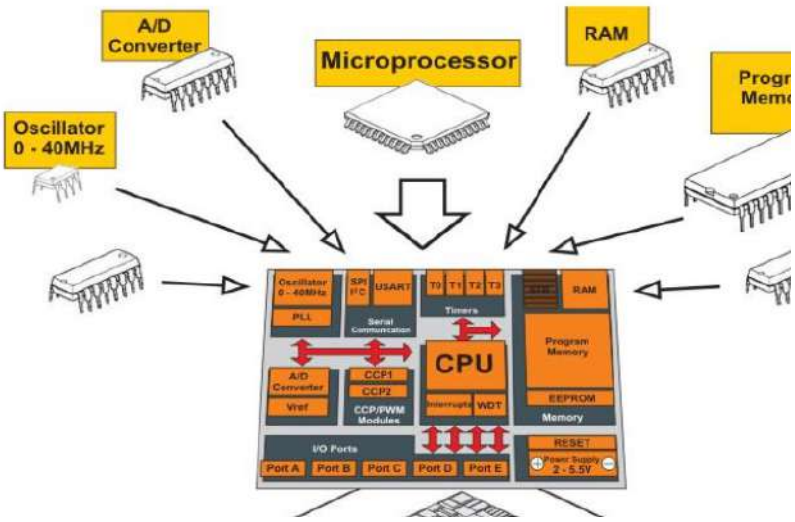


Figure I. 12 - Simplified schematic of the intelligent urban lighting system and the developed prototype.

The main role of a microcontroller is to execute specific, real-time control tasks with precision and reliability. It continuously acquires data from sensors, processes it according to programmed control algorithms, and actuates the corresponding outputs, such as regulating temperature, controlling motor speed, and triggering safety mechanisms.

Unlike microprocessors, microcontrollers generally operate without a full operating system, relying instead on directly programmed loops or lightweight real-time operating systems (RTOS). This allows deterministic behavior, a key requirement for time-sensitive applications.

Their importance is fundamental across a wide range of domains, including industrial automation, automotive embedded systems (ABS, airbag systems, electronic fuel injection), medical devices, IoT (Internet of Things), and robotics. By combining computational capability, energy efficiency, and compact design, microcontrollers enable the creation of intelligent, autonomous, and highly optimized electronic systems.

I.3.3 Differences Between a Microcontroller and a Microprocessor

Although microcontrollers and microprocessors share a similar core principle — executing instructions to process data — they differ significantly in architecture, functionality, and applications. These distinctions determine their suitability for specific types of embedded or general-purpose systems.

A) Architecture:

A microprocessor is primarily designed for high-speed computation. It lacks integrated memory and peripherals, relying on external components such as RAM, ROM, and I/O controllers to operate. This modular architecture provides high flexibility and scalability but results in greater hardware complexity and power consumption. Conversely, a microcontroller follows a System-on-Chip (SoC) architecture, integrating the processor, memory, and input/output peripherals within a single chip. This integration minimizes energy consumption, reduces size and cost, and simplifies system design. For example, an Intel Core i7 operates with multiple cores clocked at several gigahertz and large cache memories, making it ideal for multitasking and computationally intensive tasks. In contrast, a PIC18F452 microcontroller runs at about 40 MHz with a few tens of kilobytes of memory but efficiently performs precise control operations in embedded applications.

B) Functionalities:

A microprocessor is designed to run complex operating systems such as Windows or Linux, handling multiple applications simultaneously. It manages virtual memory, cache

hierarchies, and vector instruction sets, enabling efficient execution of scientific computations, AI algorithms, and multimedia processing.

A microcontroller, on the other hand, focuses on real-time control and hardware interaction. It includes dedicated modules such as watchdog timers to prevent software crashes, PWM (Pulse Width Modulation) units for motor control, and built-in communication interfaces (SPI, I²C, UART, CAN) for direct connection with sensors and actuators.

C) Typical Applications:

Microprocessors are typically used in computers, servers, and high-performance systems requiring intensive computation — such as AI processing, scientific simulations, and video rendering.

Microcontrollers are primarily used in embedded systems, where reliability, real-time operation, and low power consumption are essential. They power devices such as drone control boards, mobile robots, smart home systems, IoT devices, and energy controllers in smart grids.

In summary, the microprocessor dominates environments where performance and multitasking are priorities, while the microcontroller excels in embedded applications requiring compactness, autonomy, and deterministic real-time control.

I.3.4 Defining the Specifications for Selecting a Processor or Microcontroller

Selecting the appropriate processor or microcontroller is a critical step in the design of an embedded system, as it determines the overall performance, cost, and scalability of the final product. This choice must be guided by a well-defined specifications document (cahier des charges) that outlines the system's objectives, constraints, and operational requirements. The hardware and software architecture of the embedded platform will ultimately depend on this initial decision.

The synthesis diagram highlights two major scenarios. When a project must manage multiple applications simultaneously or when the requirements are not fully defined, a microprocessor is the preferred choice due to its expandable memory, high processing power, and scalability. Conversely, for a dedicated, application-specific project, a microcontroller is better suited, as it integrates the processor, memory, and I/O peripherals within a single compact and energy-efficient chip.

In order to establish the specifications for choosing the right component, several selection criteria must be evaluated:

- Processing Capacity: The computational performance required is a decisive factor. A microprocessor is ideal when the system must handle complex algorithms, multitasking, or real-time data processing at high speed. Conversely, for applications with well-defined and repetitive tasks such as control, regulation, or monitoring, a microcontroller is sufficient and more cost-effective.
- Integrated Peripherals: The level of integration is another major consideration. Microcontrollers typically include embedded peripherals such as ADC/DAC converters, timers, serial communication interfaces (UART, SPI, I²C), and PWM modules, minimizing the need for external components. Microprocessors, however, require additional hardware but support advanced operating systems such as Linux, Windows Embedded, or Android, offering greater flexibility in software development.
- Reliability and Safety: In safety-critical applications such as aerospace, medical devices, or automotive electronics, certified microcontrollers designed for functional safety standards (e.g., ISO 26262, IEC 61508) are often preferred. In contrast, microprocessors are more common in high-performance environments requiring extensive software capabilities, such as industrial automation or embedded AI systems.

A well-structured requirements document should therefore balance these parameters — performance, integration, reliability, power consumption, and cost — to guide the optimal selection between a microcontroller and a microprocessor.

I.3.5 Specific Project Requirements

The choice between a microcontroller and a microprocessor depends largely on the specific needs of the application. In industrial control systems, where real-time monitoring of sensors and actuators is essential, the microcontroller is naturally favored due to its low latency, simplicity, and direct hardware control capabilities. Conversely, in multimedia or high-performance applications—such as computer vision systems, smart devices, or connected objects requiring complex data handling—the microprocessor becomes indispensable. Its ability to manage rich graphical interfaces, large databases, and complex communication protocols makes it the preferred option for such demanding environments.

Flexibility is another determining factor. When a project must evolve rapidly, allow for frequent software updates, or operate across diverse environments, the microprocessor provides the necessary scalability and adaptability. On the other hand, if the objective is to develop a robust, stable, and application-specific solution, the microcontroller remains the most suitable choice thanks to its reliability and predictable behavior.

I.3.6 Power, Cost, and Size Constraints

Hardware constraints often play a decisive role in system design. In battery-powered or portable applications, power consumption becomes a critical parameter. Microcontrollers, with their low-power modes and optimized energy efficiency, are typically preferred to extend battery life. Microprocessors, being more power-hungry, require additional thermal management and stable power supplies, which increases system complexity.

Production cost is another key consideration, especially for mass-produced systems. Microcontrollers, by integrating most peripherals on a single chip, reduce the need for external components, resulting in a lower overall cost. Microprocessors, in contrast, entail higher expenses—not only for the component itself but also for the external memory, I/O controllers, and supporting hardware required for their operation.

Finally, size and compactness strongly influence the choice. Microcontrollers, which integrate memory, processing, and interfaces in a single compact package, are ideal for miniaturized embedded applications such as wearable or portable devices. Microprocessors, being more complex and often mounted on multi-chip boards, are better suited to systems where space is less constrained and performance is prioritized.

In summary, the final decision must balance application requirements, energy constraints, production costs, and system size to achieve the optimal design for an embedded solution.

I.4 Presentation of Some Microcontroller Boards

Development boards are essential platforms for embedded systems designers. They facilitate programming, rapid prototyping, and functional validation of applications before their final integration. They allow direct hardware manipulation, testing of algorithms, and easy interfacing with various sensors, actuators, and communication modules.

Among the most popular boards used in education, research, and industry, the following are widely adopted: Arduino, Raspberry Pi, and STM32 Discovery Board.

Each offers distinct technical features that meet specific needs in terms of computing power, flexibility, and cost — ranging from simple educational projects to advanced high-performance embedded systems.

I.4.1 Arduino

The Arduino board is an open-source prototyping platform widely known for its simplicity and flexibility. It is based on ATmega microcontrollers (8-bit AVR architecture) and provides a user-friendly development environment (Arduino IDE), suitable for both beginners and experienced engineers. Main features (Arduino Uno):

- Microcontroller: ATmega328P (AVR RISC, 8-bit)
- Clock frequency: 16 MHz
- Memory: 32 KB Flash, 2 KB SRAM, 1 KB EEPROM
- I/O pins: 14 digital (6 PWM) and 6 analog inputs
- Communication interfaces: UART, SPI, I²C
- Operating voltage: 5 V

Thanks to its wide community and rich library ecosystem, Arduino is ideal for basic robotics, IoT devices, environmental sensors, and educational prototypes.

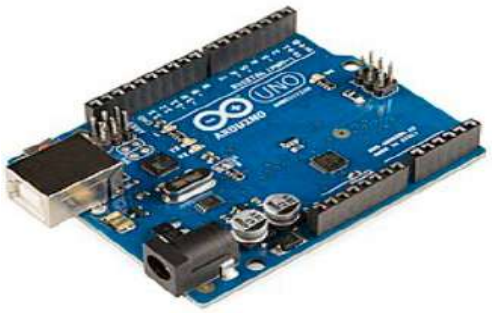


Figure I. 13 - Arduino Uno development board based on the ATmega328P microcontroller

I.4.2 Raspberry Pi

Unlike Arduino, the Raspberry Pi is not a simple microcontroller board but a single-board nano-computer capable of running a full operating system (Linux, Raspbian, Ubuntu). It is powered by an ARM processor and provides significantly higher performance, making it suitable for applications requiring intensive computation, advanced connectivity, or graphical interfaces. Main features (Raspberry Pi 4 Model B):

- Processor: ARM Cortex-A72 (quad-core, 1.5 GHz)

- RAM: 2 GB, 4 GB, or 8 GB LPDDR4
- Storage: microSD card
- Interfaces: 40 GPIO, HDMI, USB 2.0/3.0, Ethernet, Wi-Fi, Bluetooth
- Operating system: Linux-based (supports Python, C/C++, Java, etc.)

This board is particularly suitable for computer vision, IoT gateways, embedded servers, and intelligent robotics applications.

Figure I. 14 - Raspberry Pi 4 Model B — a single-board nano-computer

I.4.3 STM32 Discovery Board

The STM32 Discovery Board, developed by STMicroelectronics, is a professional prototyping platform designed for real-time embedded applications.
It is based on ARM Cortex-M microcontrollers, known for their low power consumption and high performance. Main features (STM32F407 Discovery):

- Microcontroller: STM32F407VGT6 (ARM Cortex-M4, 32-bit)
- Clock frequency: up to 168 MHz
- Memory: 1 MB Flash, 192 KB SRAM
- Interfaces: USART, SPI, I²C, CAN, USB OTG, Ethernet (depending on model)
- Integrated peripherals: accelerometer, 12-bit ADCs, advanced timers
- Operating voltage: 3.3 V

Thanks to its powerful processor and integrated modules, this board is ideal for industrial control systems, automation, real-time robotics, and complex data acquisition applications.

Figure I. 15 - STM32F4 Discovery Board featuring an ARM Cortex-M4 microcontroller

I.5 How to Select and Integrate a Development Board into an Embedded Project

The selection and integration of a development board is a strategic step in the design of an embedded system. This decision depends on functional requirements, technical and economic constraints, and the educational or industrial objectives of the project. A well-chosen board ensures not only performance and reliability but also ease of implementation and scalability of the prototype.

I.5.1 Project Requirements Assessment

The first step is to analyze the specific needs of the project in order to identify the most suitable board. This assessment is based on the following criteria:

- Nature of processing tasks: distinguish between simple tasks (sensor reading, actuator control) and complex ones (image processing, machine learning, IoT communication, etc.).
- Peripherals to interface: number and type of sensors, communication modules (UART, SPI, I²C, CAN, USB, Ethernet), or output devices.
- Required processing speed and power: depending on the algorithms to be executed or the system's real-time responsiveness.
- Available memory: amount of RAM for data, Flash for program storage, and EEPROM for saved parameters.

This analysis helps determine whether a low-cost, easy-to-use board (such as Arduino Uno) is sufficient, or whether a more powerful and connected platform (such as Raspberry Pi or STM32 Discovery) is required.

I.5.2 Hardware and Software Compatibility

Once the project needs are defined, it is essential to verify the board's compatibility with the project's hardware and software environment.

Hardware compatibility:

- Availability and number of GPIO pins.
- Support for required communication protocols (SPI, I²C, UART, CAN, etc.).
- Ability to connect additional modules or shields.

Software compatibility:

- Availability of development environments (IDEs) and supported programming languages.
- Access to libraries, frameworks, and example projects.
- Compatibility with embedded operating systems (Linux, FreeRTOS, etc.).
- Scalability: possibility to add new sensors or functionalities without redesigning the entire system.

I.5.3 Economic Considerations

Cost is a major factor, particularly for educational projects or systems intended for mass production. Key elements to consider include:

- Board price: Arduino boards are low-cost, while Raspberry Pi and STM32 boards are more expensive but offer higher performance.
- Required accessories: sensors, power supplies, extension boards, communication modules, etc.
- Development and maintenance cost: ease of learning, documentation availability, and software tool accessibility.
- Performance-to-cost ratio: avoid overdimensioning the hardware for simple applications.

I.5.4 Programming and Development

Development efficiency largely depends on the availability of software tools and technical support resources.

Integrated Development Environments (IDE):

- Arduino IDE: simple and intuitive, ideal for learning.
- STM32CubeIDE: professional and comprehensive, suited for real-time embedded projects.

- Linux/Python (Raspberry Pi): powerful and flexible for connected or AI-based applications.

Supported programming languages:

- Arduino → C/C++.
- Raspberry Pi → Python, C/C++, Java.
- STM32 → C/C++, Assembly, sometimes MicroPython.

Community and technical support: Open-source platforms like Arduino and Raspberry Pi benefit from vast user communities, detailed documentation, and a wealth of educational resources.

Simulation and testing: The ability to simulate algorithms before deployment (e.g., using Proteus, Tinkercad, or STM32CubeMX) facilitates project validation and debugging, improving development efficiency.

Chapter II: Processor Architectures

II Processor Architectures

The architecture of a processor refers to the internal organization of its components and the way they interact to execute instructions. It defines the relationship between the processor, memory, and peripherals, and directly influences the speed, energy efficiency, and programming complexity of an embedded system. Two major families dominate modern designs: the Von Neumann architecture and the Harvard architecture.

II.1 Neumann Architecture

The Von Neumann architecture is based on a single memory space that stores both the program instructions and the data being processed. The processor communicates with this memory through a single bus, meaning that instructions and data share the same communication path. This organization offers the advantages of simplicity and low implementation cost. It is particularly suitable for systems requiring high software flexibility, such as personal computers and workstations.

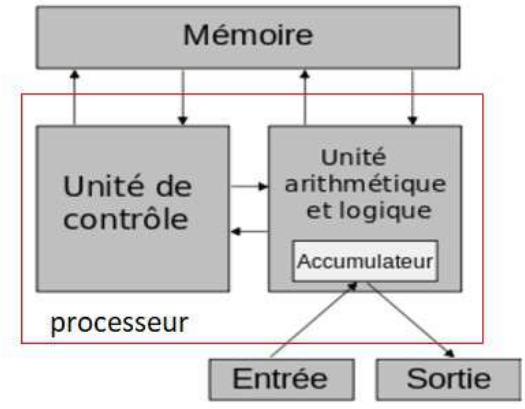


Figure II. 1 . Von Neumann Architecture

However, this architecture is constrained by a significant limitation known as the **Von Neumann bottleneck,** as the processor can access only one piece of information at a time—either an instruction or data—which slows down execution speed.

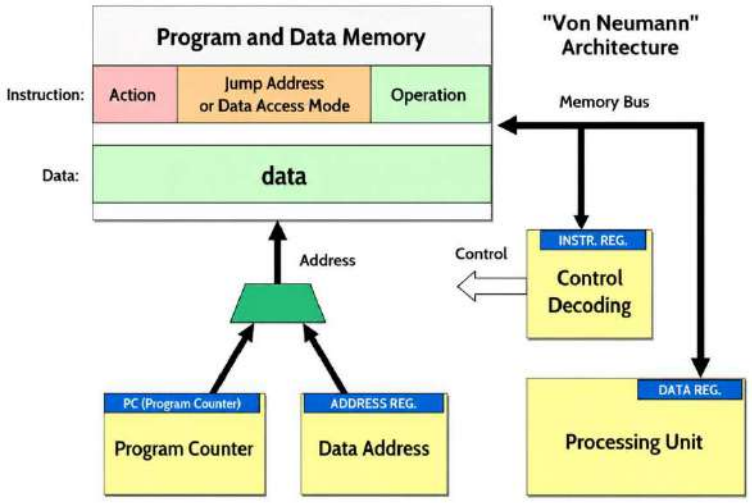


Figure II. 2 - Instruction Execution in a Von Neumann Architecture

In this architecture, the execution of a program follows the classical instruction cycle, often referred to as the **Von Neumann cycle**, which includes several successive stages:

- **Fetch (Instruction Read):** The processor reads the instruction to be executed from memory. Since there is only one bus, it can fetch only one instruction at a time.
- **Decode:** The instruction is analyzed by the control unit to identify the operation to be performed and the resources required.
- **Fetch Operand (Data Read):** If the instruction requires data, these are retrieved from memory via the same bus. This may introduce additional latency, as instructions and data share the same access path.
- **Execute:** The arithmetic and logic unit (ALU) or another component of the processor executes the requested operation.
- **Write Back:** The result is written back into memory or a register, again using the single bus.

This cycle clearly illustrates the main limitation of the Von Neumann architecture: since instructions and data cannot be transferred simultaneously, each stage must wait for bus availability. This results in reduced performance, especially in applications that require high data throughput.

II.2 Harvard Architecture

The **Harvard architecture** introduces a physical separation between the **instruction memory** and the **data memory**. The processor therefore uses **two distinct buses** — one dedicated to instructions and another to data. This organization enables the processor to fetch an instruction while simultaneously accessing data, which greatly increases execution speed and enhances parallelism.

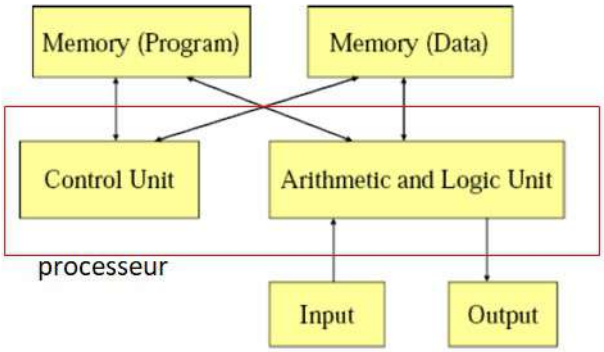


Figure II. 3 - Harvard Architecture

This structure is widely adopted in modern **microcontrollers** and **embedded systems**, where performance, energy efficiency, and responsiveness are essential. It also allows for better exploitation of **pipelining**, a technique that divides instruction execution into parallel stages. However, this design is more complex and slightly more expensive to implement than the Von Neumann architecture.

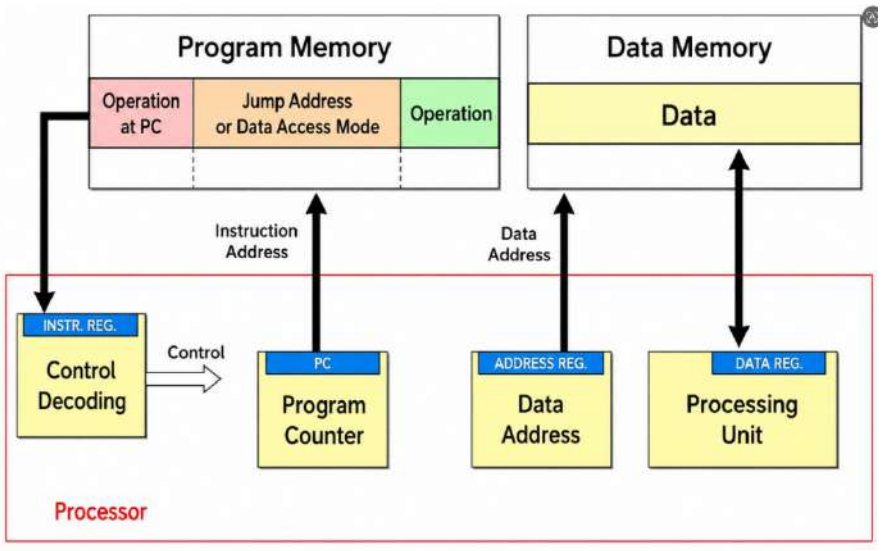


Figure II. 4 - Instruction Execution in a Harvard Architecture

The instruction cycle in a Harvard architecture is optimized by the presence of two separate memories and two buses, allowing faster and more efficient execution since multiple operations can occur in parallel:

- **Fetch (Instruction Read):** The processor retrieves the instruction from the program memory through the instruction bus.
- **Decode:** The instruction is interpreted by the control unit to determine the operation and required resources.
- **Fetch Operand (Data Read):** Simultaneously, using the separate data bus, the processor accesses the necessary data without waiting for the instruction read to complete.
- **Execute:** The arithmetic logic unit (ALU) or another processing unit performs the operation.

Write Back: The result is stored in data memory or a register using the dedicated data bus, while a new instruction can already be fetched.

This simultaneous operation significantly reduces idle time and enables pipeline organization, where several instructions are processed at different stages of execution. For instance, while one instruction is being executed, another can be decoded and a third fetched from memory. This approach eliminates the Von Neumann bottleneck and provides a substantial improvement in both processing speed and overall system throughput.

II.3 Comparison Between Von Neumann and Harvard Architectures

The Von Neumann architecture is characterized by its simplicity of design and flexibility of use. It relies on a single shared memory and one common bus for both instructions and data. This organization reduces hardware costs and simplifies programming, making it suitable for general-purpose computing. However, it suffers from a major limitation known as the Von Neumann bottleneck, which results from the processor's inability to access instructions and data simultaneously. As a consequence, execution speed can be significantly reduced, especially in applications requiring high-speed or intensive data processing.

In contrast, the Harvard architecture introduces a physical separation between instruction memory and data memory, each with its own dedicated bus. This design enables parallel

access to both instructions and data, eliminating the bottleneck and providing a substantial increase in processing speed. Additionally, it improves energy efficiency, making it particularly well-suited for embedded systems, real-time applications, and digital signal processing (DSP), where performance and resource optimization are critical.

Table II. 1- Comparison Between Von Neumann and Harvard Architectures

Characteristic	Von Neumann Architecture	Harvard Architecture
Memory Organization	Single memory for both instructions and data	Separate memories for instructions and data
Bus Configuration	Single shared bus	Distinct buses for instructions and data
Access to Instructions/Data	Sequential (one access at a time)	Parallel (simultaneous access possible)
Design Complexity	Simpler, lower cost	More complex, slightly more expensive
Execution Speed	Slower due to bottleneck	Faster, eliminates the bottleneck
Flexibility	High (easier programming and design)	More specialized
Typical Applications	General-purpose computers and systems	Embedded systems, DSPs, real-time applications
Energy Consumption	Higher under intensive workloads	More energy-efficient for targeted processing

II.4 Choosing Between Von Neumann and Harvard Architectures

The choice between **Von Neumann** and **Harvard** architectures primarily depends on the project constraints, performance requirements, and available resources. Each architecture offers specific advantages and limitations that directly affect the design and implementation of an embedded system.

II.4.1 Selection Criteria

The main criteria to consider when selecting an architecture include:

- **System Complexity:** The Von Neumann architecture is simpler to design and program, while the Harvard model requires more rigorous management of the separation between instructions and data.

- **Expected Performance:** For applications demanding high-speed or parallel processing, the Harvard architecture is more suitable. For general-purpose systems, Von Neumann remains sufficient.
- **Power Consumption:** Harvard is generally more energy-efficient, particularly in real-time and DSP systems, whereas Von Neumann tends to consume more power during intensive operations.
- **Cost and Accessibility:** Von Neumann architectures are more economical in terms of hardware design, while Harvard requires additional resources such as separate memory blocks and data buses.
- **Ease of Development:** Von Neumann simplifies programming and system integration, making it an ideal choice for prototypes, educational systems, and entry-level embedded projects.

II.4.2 Specific Applications for Each Architecture

The decision between Von Neumann and Harvard architectures is not merely theoretical; it directly influences the target application domains where each model proves most effective. The fundamental difference — a shared memory and bus in Von Neumann versus separate instruction and data paths in Harvard — naturally determines their areas of use. Some applications favor the flexibility and simplicity of Von Neumann, while others require the speed and energy efficiency of Harvard.

A) Von Neumann Architecture:

- **Personal and Laptop Computers:** Where application diversity (office software, multimedia, programming) demands flexibility.
- **General-Purpose Operating Systems (Windows, Linux, macOS):** Require architectures capable of managing multiple processes and resources simultaneously.
- **Educational Prototypes and Academic Projects:** Its simplicity in design and programming makes it an excellent foundation for learning.
- **Non-Critical Applications Where Speed Is Not a Priority:** Such as office automation tools, simple computing devices, or low-constraint embedded systems.

B) Harvard Architecture:

- **Critical Embedded Systems (Aerospace, Automotive, Robotics):** Where reliability, high speed, and low power consumption are key.
- **Real-Time Applications:** Such as motor control, industrial process regulation, and automated control systems.
- **Digital Signal Processing (DSP):** Used in audio, video, telecommunications, and data compression.
- **Internet of Things (IoT) Devices:** Require high energy efficiency and fast local data processing within limited hardware resources.
- **Medical and Military Systems:** Where precision, responsiveness, and reliability are crucial for safety and operational performance.

II.5 Overview of Microcontroller Boards by Architecture

The study of processor architectures is not limited to theory; it finds direct application in microcontrollers and development boards widely used in embedded systems and educational environments. Depending on the adopted architecture — Von Neumann, Harvard, or a modified hybrid version — the performance, energy efficiency, and design flexibility of the system vary significantly.

II.5.1 Microcontrollers Based on the Von Neumann Architecture

Von Neumann–based microcontrollers use a single memory space for both instructions and data. This simplifies hardware design and programming but limits execution speed due to the shared data bus.

Examples:

- x86 processors (Intel, AMD): still rely on an optimized Von Neumann model.
- 8051 family: commonly used in simple, industrial, or educational applications.
- ARM Cortex-A series: used in Raspberry Pi boards, adopting a Von Neumann approach with modern enhancements (caches, pipelines, MMU).

Associated Boards:

- Raspberry Pi (Pi 3, Pi 4): ideal for learning, IoT projects, and applications requiring a full operating system (Linux).

- Industrial x86-based boards: such as embedded PCs and industrial mini-PCs.

II.5.2 Microcontrollers Based on the Harvard Architecture

In the Harvard architecture, instruction and data memories are physically separated, enabling parallel execution. This structure enhances performance, reduces latency, and optimizes energy efficiency — features that are essential in embedded systems.

Examples:

- AVR (Atmel, now Microchip): classic Harvard architecture, e.g., Arduino Uno.
- PIC (Microchip): widely used in industrial control applications.
- DSP (Digital Signal Processors): specialized for signal processing tasks.

Associated Boards:

- Arduino Uno / Mega (AVR): popular for education and rapid prototyping.
- PICKit boards (PIC-based): used in industrial and control systems.
- DSP development boards: employed in audio, imaging, and communication systems.

II.5.3 Microcontrollers Based on Modified Architectures (Optimized Von Neumann & Modified Harvard)

Most modern microcontrollers use a hybrid architecture to combine the strengths of both models.

- Optimized Von Neumann: adds cache memory, pipelines, and memory management units (MMU) to increase execution speed despite a shared bus.
- Modified Harvard: maintains separate instruction and data memories but allows certain resources to be shared through hierarchical buses or cache layers.

Examples:

- **ARM Cortex-M (STM32, nRF52, etc.):** modified Harvard architecture, ideal for embedded and IoT systems.
- **ESP32 (Espressif):** modified Harvard, integrating **Wi-Fi** and **Bluetooth** connectivity.

- **STM32 Discovery / Nucleo boards:** educational and industrial boards based on ARM Cortex-M.
- **Raspberry Pi:** optimized Von Neumann with ARM Cortex-A, suited for complex systems.

Table II. 2 - Summary Table

Architecture	Main Characteristics	Examples of Microcontrollers	Associated Boards
Von Neumann	Single memory, shared bus. Simple design but limited by bottleneck.	8051, ARM Cortex-A, x86	Raspberry Pi, industrial x86 boards
Harvard	Separate instruction/data memory, parallel execution, higher speed.	AVR, PIC, DSP	Arduino Uno/Mega, PICKit boards, DSP boards
Modified Harvard	Combines flexibility and speed; uses pipelines, caches, and energy optimization.	ARM Cortex-M (STM32), ESP32	STM32 Nucleo/Discovery, ESP32 DevKit
Optimized Von Neumann	Adds cache, pipeline, MMU, hierarchical bus. Improves flexibility and performance.	ARM Cortex-A, modern x86 processors	Raspberry Pi 3/4, embedded PCs

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.

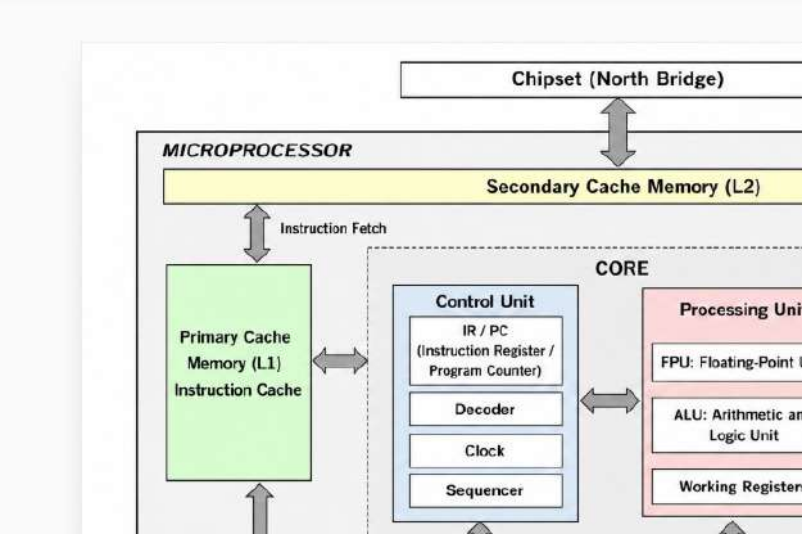


Figure III. 1 - Physical architecture of a single-core processor with its main functional units.

- **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.

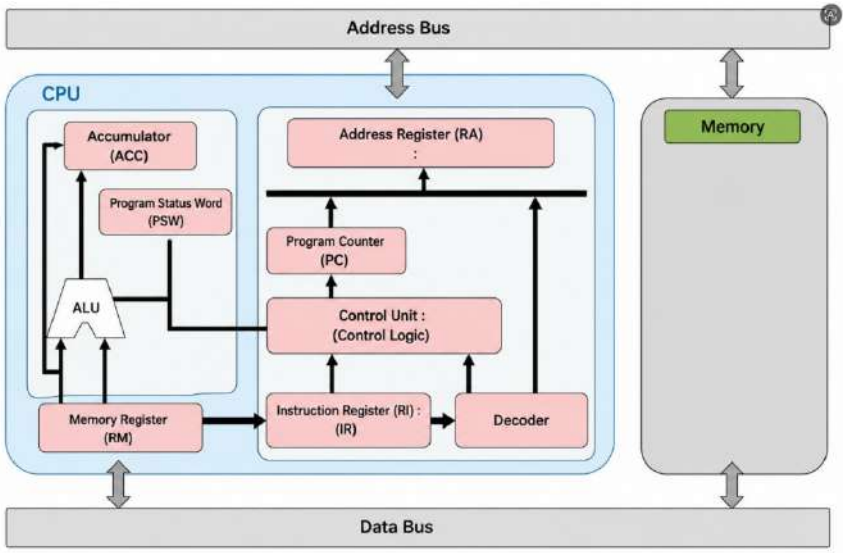


Figure III. 2 - Internal organization of processor registers and their interaction with the ALU, CU, and memory.

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).

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.

C) OR Gate

- Function: Produces 1 if at least one input is 1.
- Its activation also depends on the control 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: **S = A ⊕ B** (using XOR gate).
    - Carry: **C = A • B** (using AND gate).
- **Full Adder:** Adds three bits (A, B, and carry-in, Cin).
    - Sum: **S = A ⊕ B ⊕ Cin**
    - Carry-out: **Cout = (A • B) + (Cin • (A ⊕ 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

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)

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	Flip-flops change state	Simple, slower due to	Frequency division, low-speed

Type of Counter	Operating Principle	Main Characteristics	Typical Applications
Counter (Ripple)	sequentially; only the first receives the clock.	propagation delays.	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:

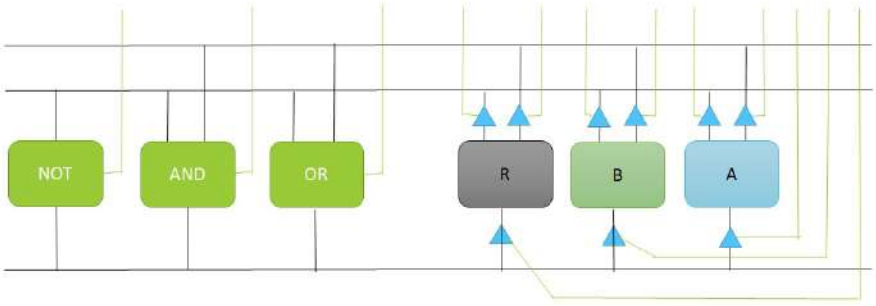
R = Not(A) And Not(B) Or A And B

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**

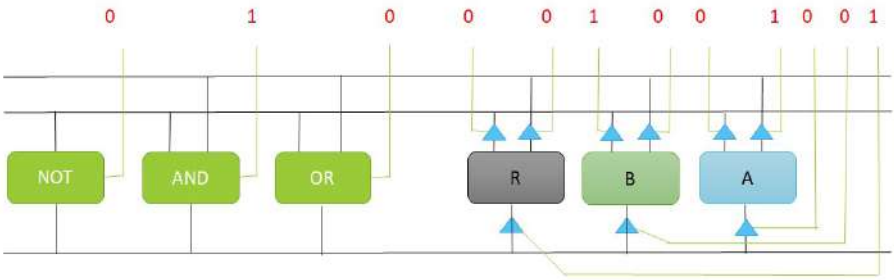
The first diagram shows a simplified architecture.

- 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.

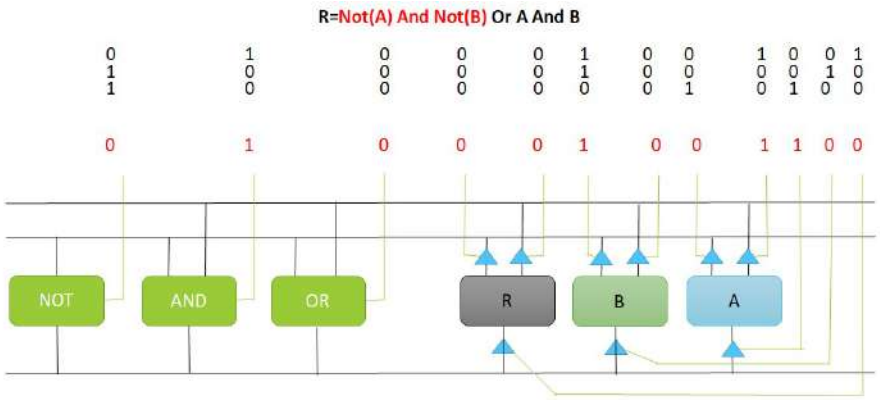


➢ Step 3 – Intermediate Operations

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)**).

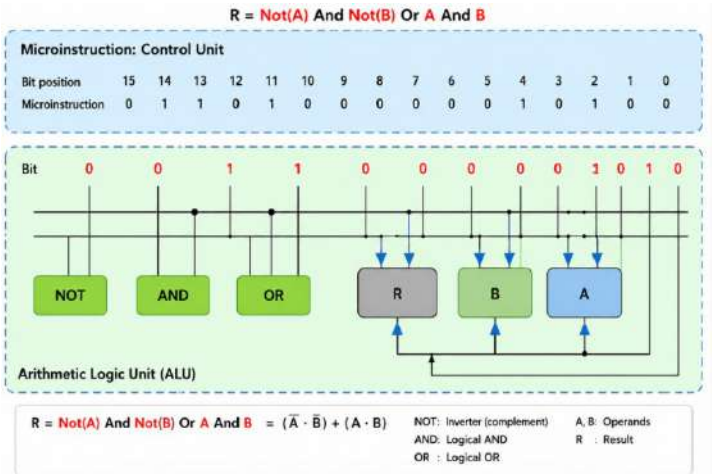
Each column of the corresponding diagram represents a combination of A and B values, showing the resulting output R.



➢ 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.

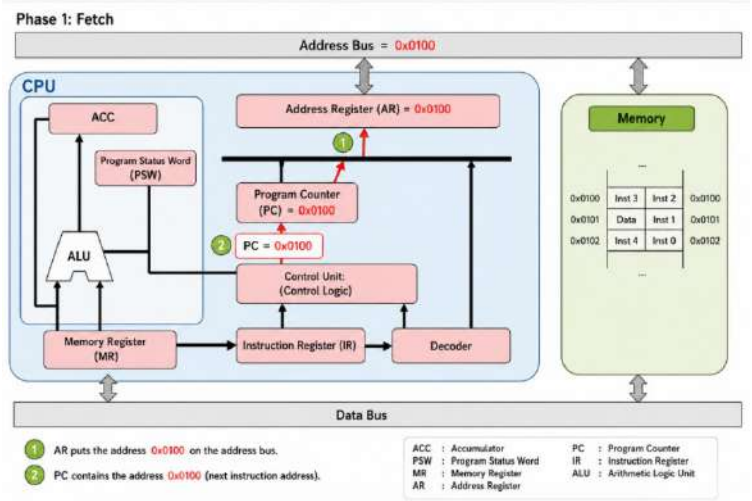
III.6.1 Fetch Phase (Instruction Retrieval)

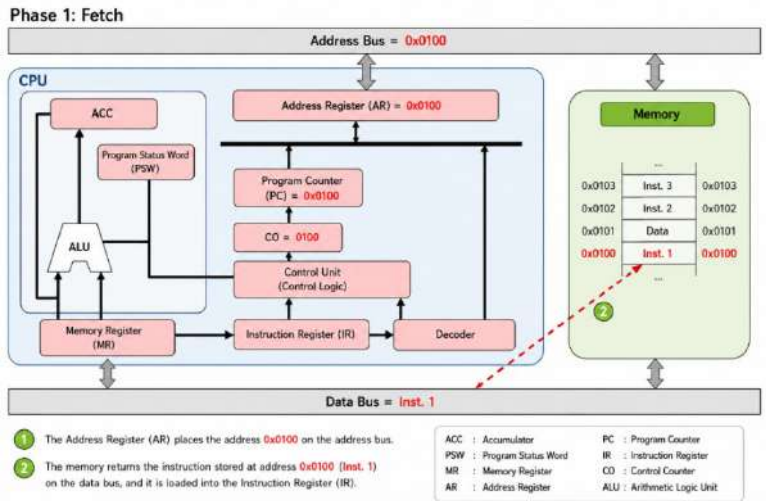
This phase consists of retrieving the instruction to be executed from memory:

- 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.





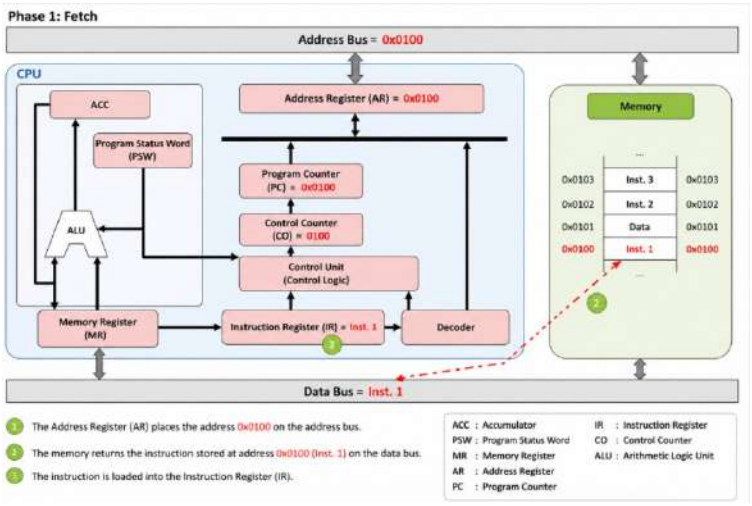


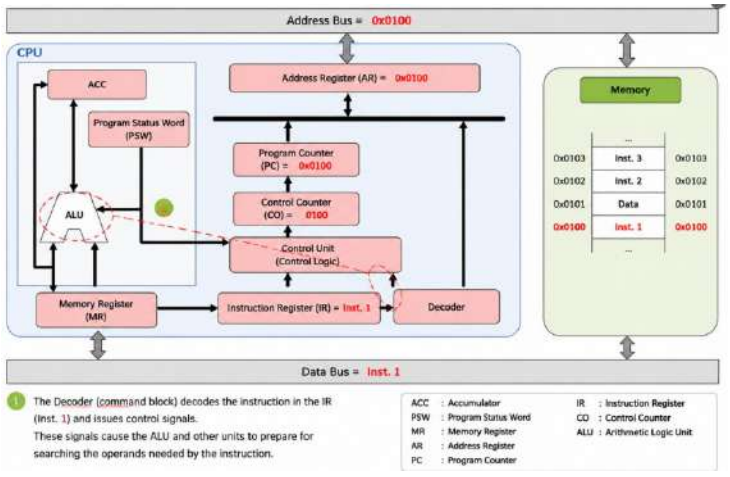
Figure III. 3 - Fetch Phase (Instruction Retrieval)

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.



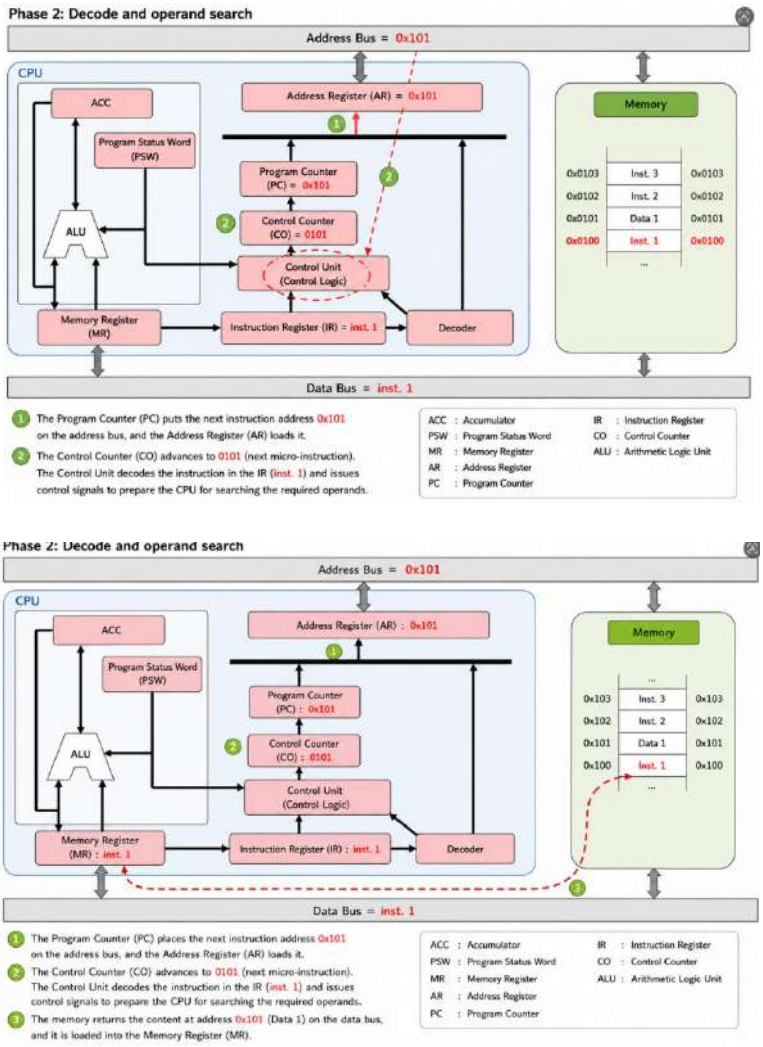


Figure III. 4 - Decode Phase (Instruction Decoding and Operand Retrieval)

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.).

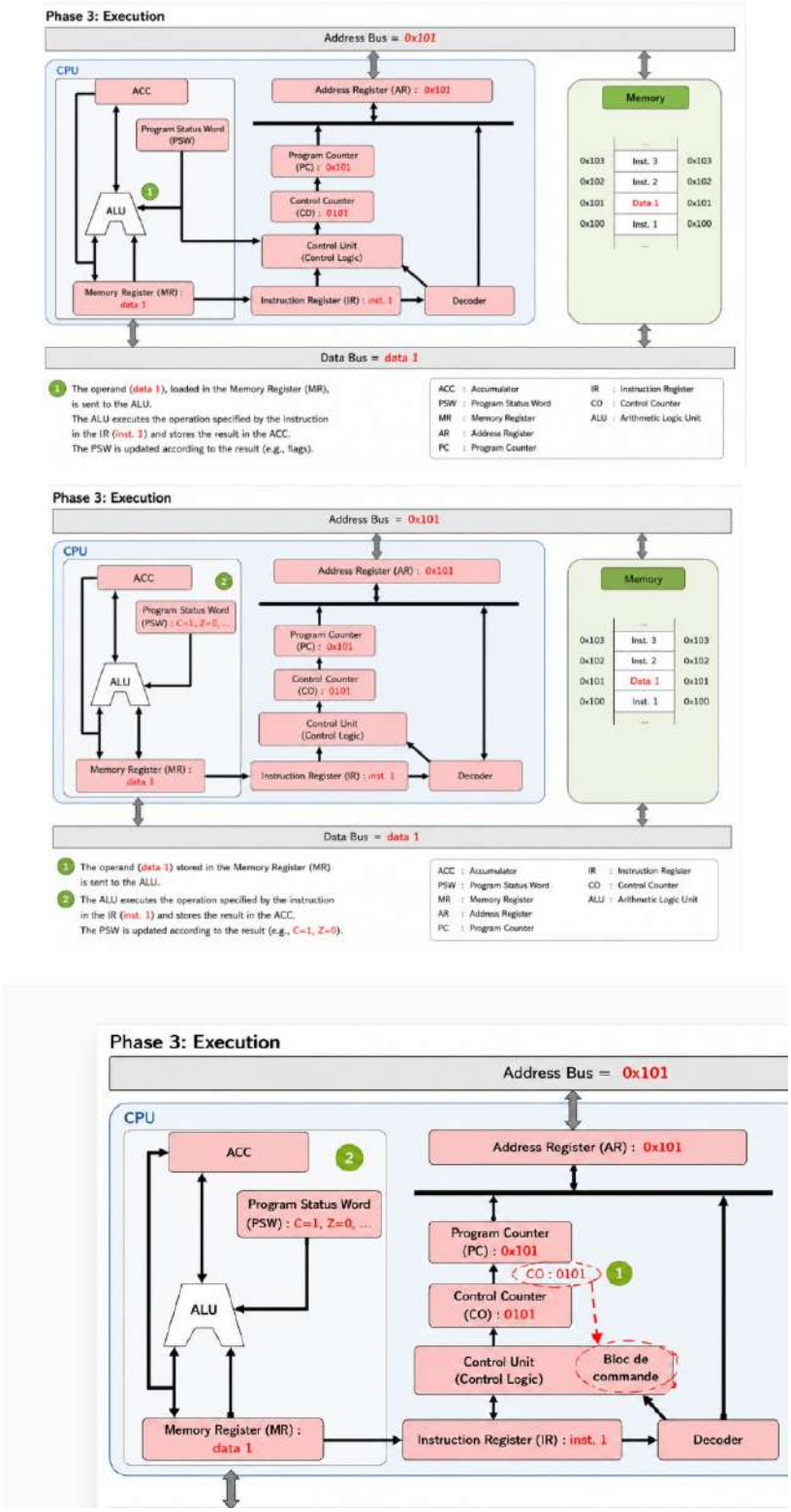


Figure III. 5 - Execute Phase (Operation Execution)

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.

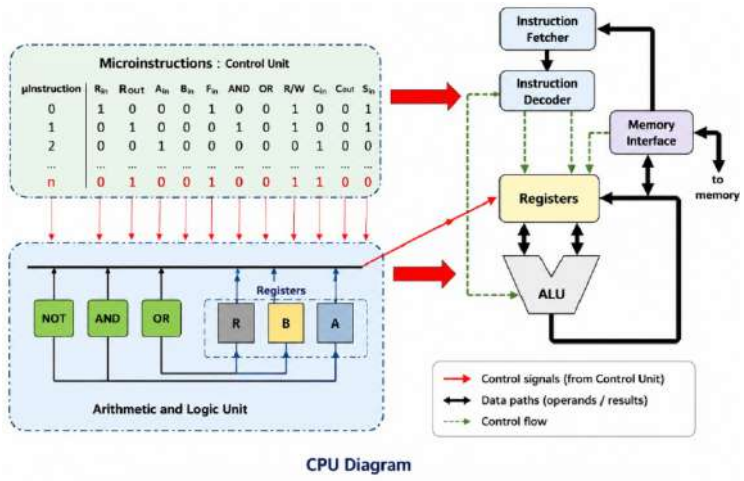


Figure III. 6 - interaction between control unit microinstructions, the ALU, and the main CPU components.

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.

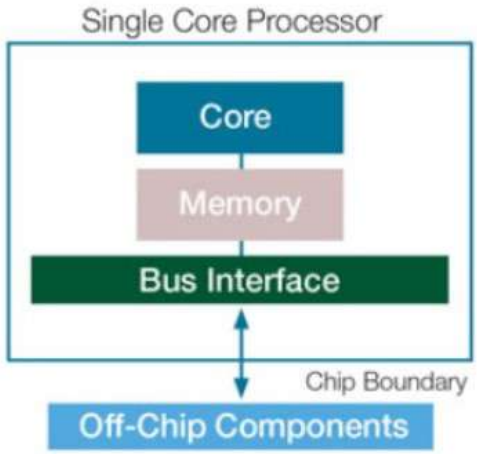


Figure III. 7 - Single Core Processor

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.

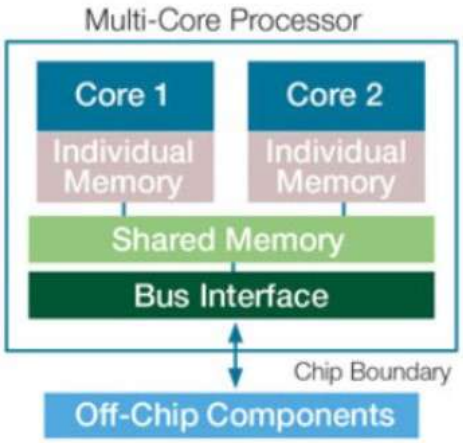


Figure III. 8 - Multi- core processor

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.

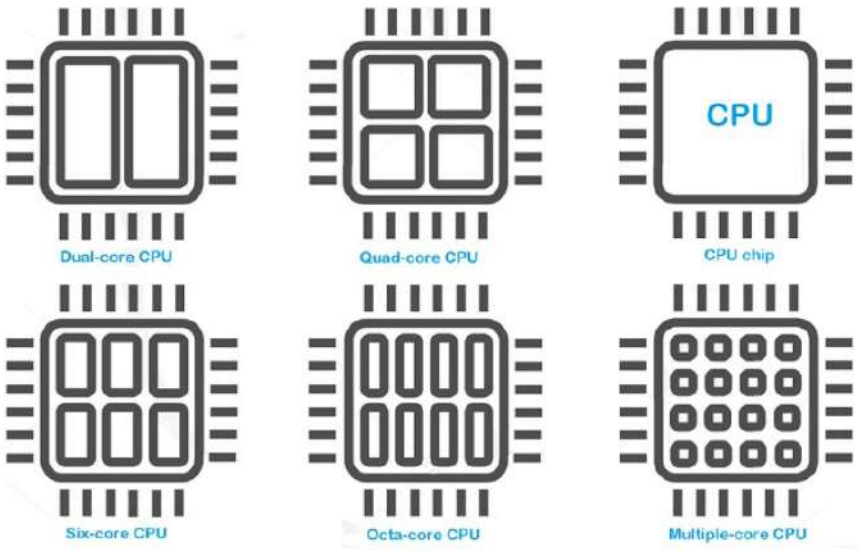


Figure III. 9 - hardware parallelism

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. 10 - 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.

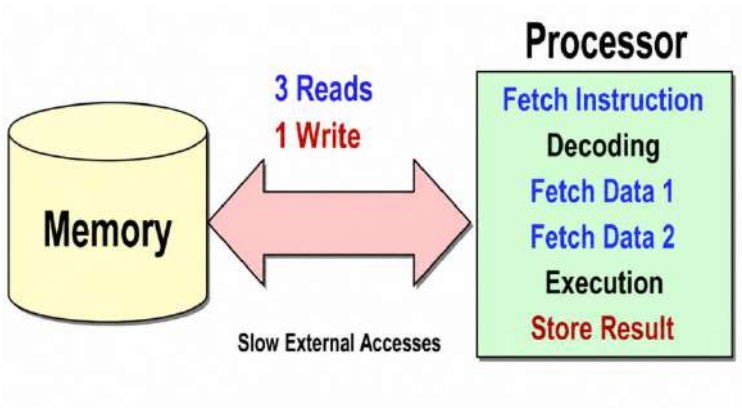


Figure III. 11 - Data and Instruction Flow Between Processor and Memory in a CISC Architecture

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.

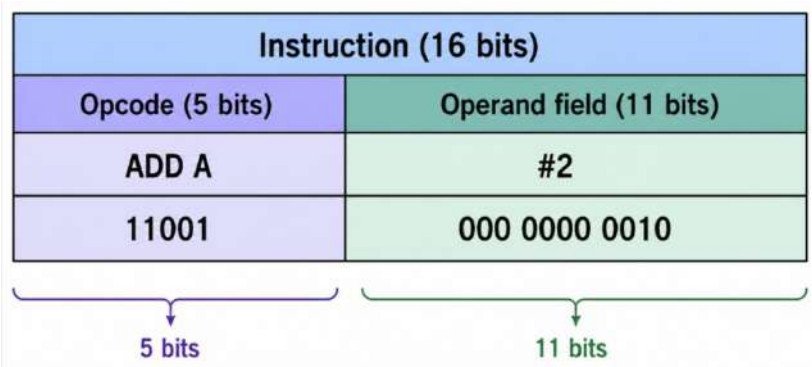


Figure III. 12 - Example of a CISC Instruction Format (16-bit Instruction Encoding: Opcode and Operand Fields)

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).

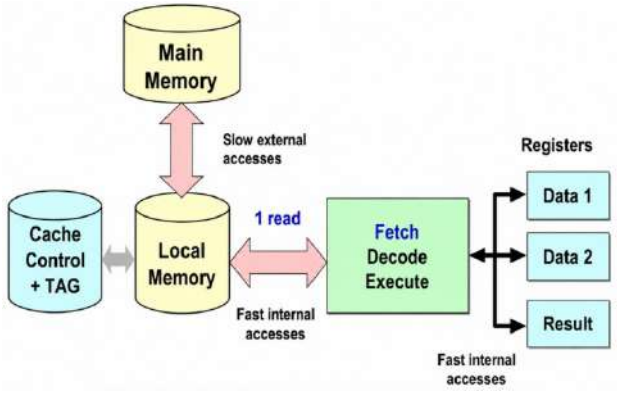


Figure III. 13 - Data Access and Execution Flow in a RISC Architecture (Using Local Memory and Register Operations)

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. 14 , 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.

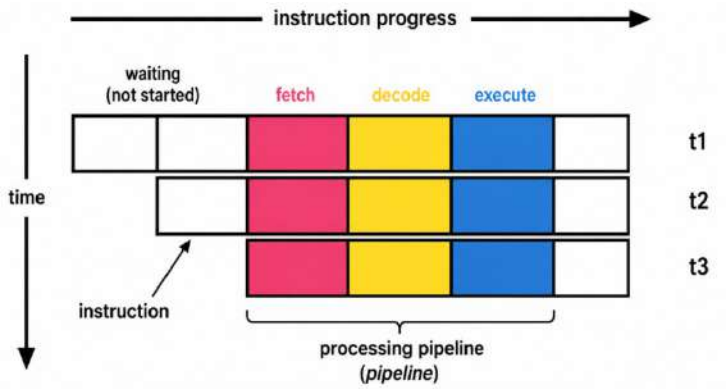


Figure III. 15 - Princile of Pipeline Operation

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.

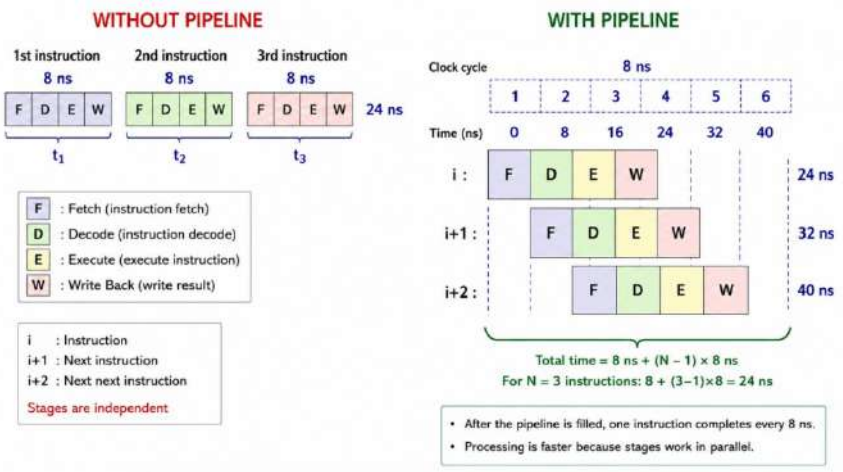


Figure III. 16 - Execution Time with and without Pipeline

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.

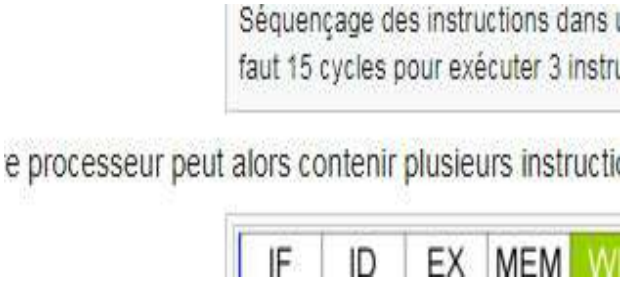


Figure III. 17 - Stages of a Classical RISC Pipeline

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.

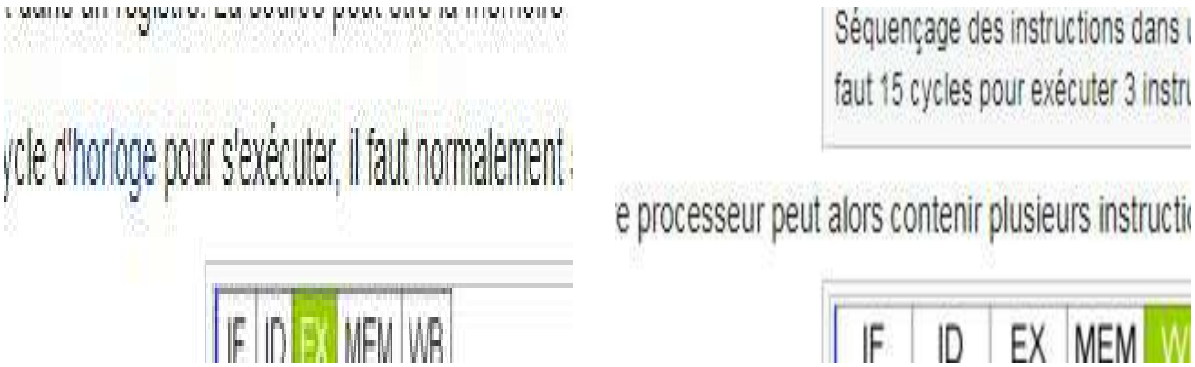


Figure III. 18 - Instruction Sequencing with and without Pipeline

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.

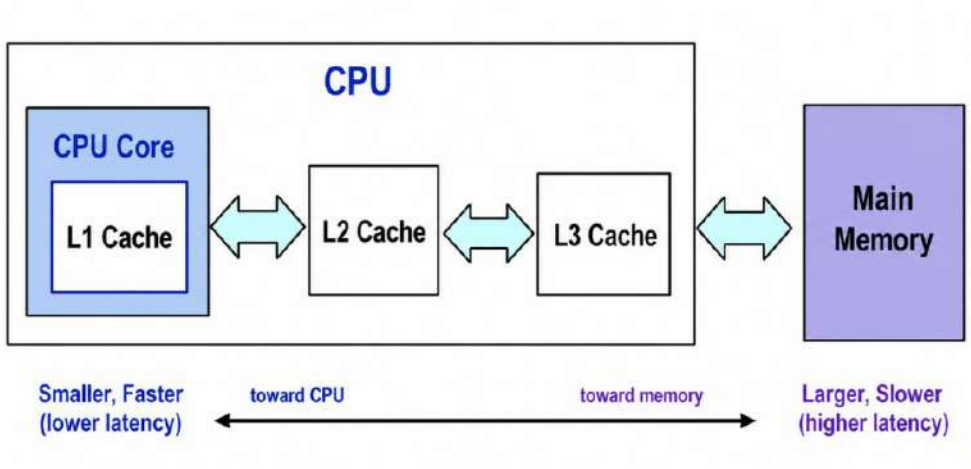


Figure III. 19 - Cache Memory Hierarchy (L1, L2, L3) and Relation with RAM

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.10.6 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.11 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.11.1 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.11.2 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.11.3 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.11.4 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.

Chapter IV: Memory architectures and technologies

IV Memoires

IV.1 Introduction to Memories

Memory is one of the fundamental components of any computing or embedded system. It plays a central role by ensuring both:

- the permanent or semi-permanent storage of programs (firmware, operating systems, applications) that define the overall behavior of the system;
- the temporary storage of data required for processing by the processor, serving as a dynamic workspace during instruction execution.

Thus, memory serves a dual purpose: it preserves essential information for system operation and provides a fast, flexible area for data processing.

Moreover, it acts as a critical interface between the processor and the rest of the system. Its access speed, storage capacity, and technological characteristics directly influence:

- overall system performance,
- power consumption,
- reliability, and
- the overall cost of the hardware solution.

In embedded systems, memory therefore goes far beyond a simple storage function. It represents a strategic component of both the hardware and software architecture, whose understanding and optimization are essential for designing efficient, reliable, and application-specific systems — particularly those constrained by size, cost, energy autonomy, or real-time requirements.

IV.2 Impact on Device Performance

Memory has a decisive impact on the performance of an embedded system. Its characteristics — access speed, capacity, and technology — directly affect the processor's efficiency in executing instructions and processing data.

A first impact concerns execution speed: if read/write operations are too slow, the processor must wait for data, reducing overall efficiency.

A second factor is storage capacity. Insufficient memory limits the size of programs and data handled. In demanding applications such as artificial vision or real-time signal processing, this limitation may compromise project feasibility.

Another crucial aspect is power consumption. In battery-powered devices — such as wireless sensors, IoT systems, or portable medical devices — memory must be optimized to reduce power usage while maintaining reliable and fast data access.

Finally, memory quality directly affects system reliability and robustness. Poorly adapted or low-quality memory can cause data loss, computation errors, or instability — unacceptable in critical applications.

IV.3 General Organization of Memory

Memory is a fundamental component of digital systems. Its organization is based on three main elements: addressing, decoding, and data management.

As illustrated in Figure 87, a memory unit consists of the following components:

- Address Bus: allows the processor to select a specific memory word. With n address lines, the memory can contain 2^n words.
- Address Decoder: receives the address input and activates a single line corresponding to the selected memory word.

- Memory Matrix: a set of cells organized into 2^n words of m bits each. Each word corresponds to a row selected by the decoder.
- Data Bus: carries information read from or written to the memory. Its width is m bits, matching the size of one memory word.
- Control Signals (Read/Write): define the operation to be performed:
  - *Read*: reads the content of the selected word and outputs it on the data bus.
  - *Write*: writes a new word into the selected row.

Figure 87 – General organization of a memory (decoder, memory matrix, and data bus)

When an address is placed on the address bus, the decoder selects the corresponding word in the memory matrix. The control signal then determines the operation:

- **Read**: the content of the memory word is transferred through a multiplexer to the data bus and sent to the processor.
- **Write**: data from the data bus are transferred through a demultiplexer to the selected memory cell, replacing the old value.

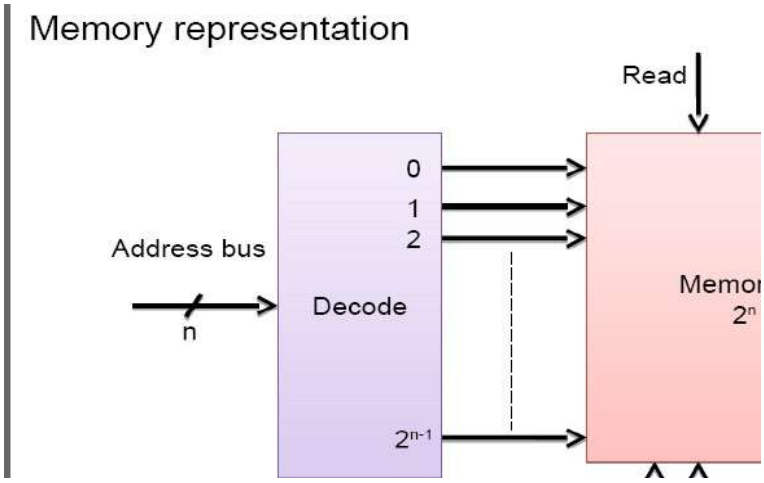


Figure VI. 1 - Read and Write process in memory

IV.4 Memory Read/Write Timing Diagram

The read/write timing diagram describes the **temporal sequence of signals** exchanged between the microprocessor and the memory to ensure correct data transfer. Each operation (read or write) follows a precise, clock-synchronized sequence.

IV.4.1 Steps of a Memory Operation

1. The microprocessor first places the target memory address on the address bus.

2. The Chip Select (CS) signal is activated to indicate that the addressed memory component should respond.
3. Depending on the operation:
    - For a read, the RD (Read) signal is activated, and the memory places the corresponding data on the data bus.
    - For a write, the WR (Write) signal is activated, and the data provided by the processor are stored in the addressed memory cell.
4. The time required for the memory to provide or record the data is called access time (Taccess).
5. The total duration of the operation, including signal setup and data stabilization, defines the memory cycle time (Tcycle).

IV.4.2 Timing Diagram

The figure illustrates the sequence of control and data signals:

- The address bus specifies the target memory position (@x).
- The CS signal enables the memory.
- The R/W signal defines the operation type (read or write).
- The data bus (BUS D) carries the exchanged information.

Two key time periods are distinguished:

- **Taccess**: the moment when data become valid;
- **Tcycle**: the duration of a full memory operation cycle.

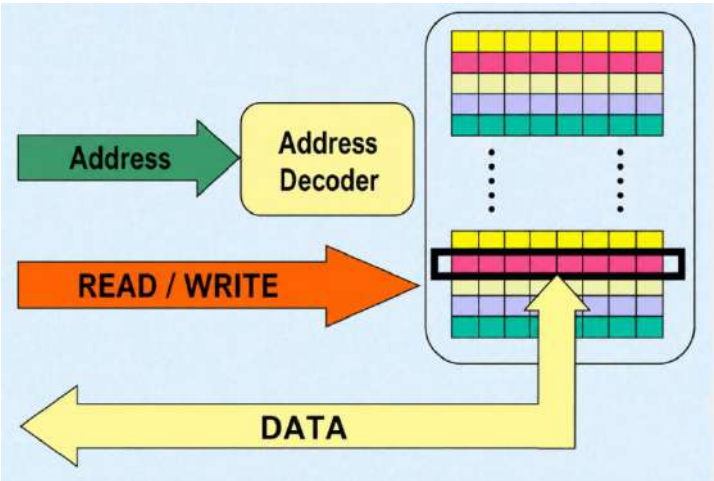


Figure VI. 2 - Memory Read/Write Timing Diagram

Respecting the timing diagram is crucial to ensure:

- proper synchronization between processor and memory,
- stable and reliable data exchange, and
- optimized performance (since long access times slow down the entire system).

In modern architectures, several techniques are used to reduce latency, such as:

- pipelined memory access (anticipating the next data read), and
- cache memories (L1, L2, L3), which store frequently used data to minimize slow main-memory accesses.

IV.5 Memory: Read/Write Timing Diagram

The read/write timing diagram describes the temporal sequence of signals exchanged between the microprocessor and the memory to ensure a correct data transfer. Each operation (read or write) follows a well-defined sequence, synchronized by the system clock.

IV.5.1 Execution of a Memory Operation

- The microprocessor first places the address of the target memory cell on the address bus.
- The Chip Select (CS) signal is activated to indicate that the addressed memory device must respond.
- Depending on the operation type:
  - For a read, the RD (Read) signal is activated, and the memory places the corresponding data on the data bus.
  - For a write, the WR (Write) signal is activated, and the data provided by the processor are stored in the addressed memory cell.
- The time taken by the memory to provide (or record) a data item is called the access time (Taccess).
- The total duration of the operation, including signal setup and data stabilization, is known as the memory cycle time (Tcycle).

IV.5.2 Timing Diagram

The figure below illustrates the sequence of signals during a memory operation:

- The **address bus** specifies the target memory position (@x).
- The **CS** signal enables the selected memory component.
- The **R/W** signal defines the type of operation (read or write).
- The **data bus (BUS D)** carries the data being exchanged.

Two important time intervals can be distinguished:

- **Taccess** – the moment when the data become valid;
- **Tcycle** – the duration of a complete memory operation cycle.

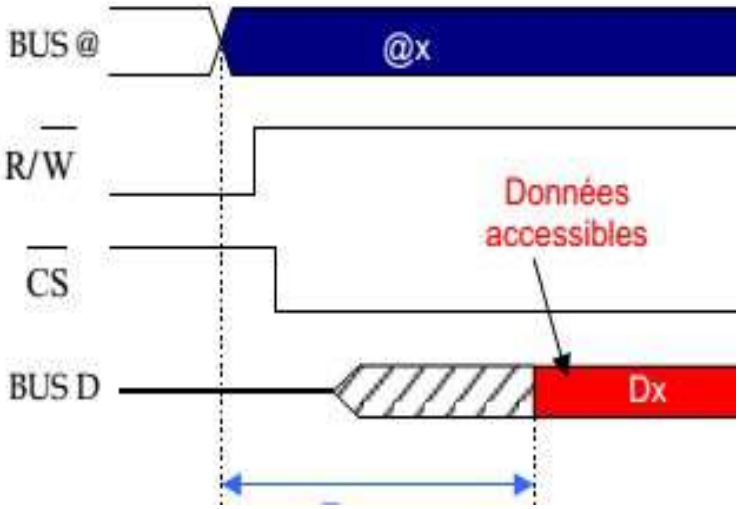


Figure VI. 3 - Memory Read/Write Timing Diagram

Respecting the timing diagram is essential to ensure:

- correct synchronization between the processor and the memory,
- stability of the data being read or written, and
- optimal system performance, since excessive access time slows down the entire system.

In modern architectures, several techniques are used to reduce this delay, such as:

- pipelined memory access (anticipating the next data read), and
- the addition of cache memories (L1, L2, L3), which store frequently accessed data to reduce slow main memory accesses.

IV.6 Memory Characteristics

The performance and role of a memory device depend on several technical parameters:

- **Capacity:**
  - Represents the size of the memory, i.e., the amount of information it can store.
  - Expressed in bits, bytes (KB, MB, GB, TB), or in the number of memory words.
  - Example: a 4 GB memory can store approximately 4 billion bytes.
- **Access Time (Taccess):**
  - The interval between a data request and its actual availability.
  - In some memories, read and write access times differ.
  - The lower the access time, the faster the memory.
- **Memory Cycle Time (Tcycle):**
  - The minimum time separating two consecutive accesses to memory.
  - Ideally close to the access time, but may include additional delays (refresh, synchronization, etc.).
- **Data Rate (or Memory Bandwidth):**
  - The amount of information transferred per second, expressed in bits/s or bytes/s.
  - Example: DDR4-3200 can reach a theoretical bandwidth of about **25.6 GB/s**.
- **Volatility:**
  - Defines whether the memory retains data when power is removed.
  - Volatile memory: RAM (data lost when power is off).
  - Non-volatile memory: ROM, Flash, EEPROM (data retained even when unpowered).

IV.7 Memory Classification

Memory occupies a central place in every embedded system. It handles the storage of programs, data, and instructions required for processor operation.
Its classification is based on several fundamental criteria:

1. Volatility: This criterion distinguishes between memories that retain information after power loss and those that lose data once power is turned off. This separation helps identify memories used for temporary storage versus those intended for permanent data retention.
2. Access Mode: Depending on their design, some memories allow sequential access (data read in a fixed order), while others support random access (direct access to any address). This feature directly affects both speed and the method of reading or writing data.

3. Functional Role: Some memories serve as working space for ongoing operations (temporary storage), while others are dedicated to permanent storage of programs and system data.
4. Technology: Memories can be built using magnetic, optical, or semiconductor technologies, depending on the desired performance, capacity, and cost.

This diversity of criteria allows memory to be grouped into two main families:

- Volatile memories, used for fast, temporary data processing.
- Non-volatile memories, designed for long-term information storage.

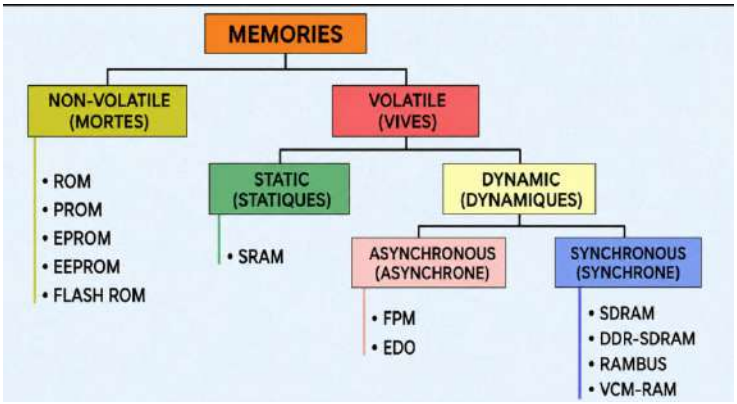


Figure VI. 4 - General Classification of Memories in Embedded Systems

IV.8 Non-Volatile Memories

Non-volatile memories are those that retain stored data even after power is turned off. They play a crucial role in embedded systems, as they store the main program (firmware), user parameters, and system boot information.

Unlike volatile memories, they do not require continuous power to preserve data, which ensures the persistence of embedded software and the stability of the device.

These memories are divided into several categories according to their structure, programming method, and rewriting capability. The main types include Read-Only Memories (ROM), mass storage memories, and Flash memories.

Read-Only Memories, or ROM, are designed to hold permanent data. They are generally programmed once and used to store the main program code or the instructions required for system initialization.

In embedded systems, ROM ensures reliable operation by maintaining a stable content, even after multiple power cycles. Different types of ROM exist depending on the programming and rewrite capabilities.

A) ROM

ROM (Read-Only Memory) is a non-volatile memory whose content is permanently defined during the manufacturing process. It is the simplest and most stable form of read-only memory. The data stored in ROM are written by the manufacturer and cannot be modified by the user. This property guarantees high reliability, since the information remains intact even without electrical power. In embedded systems, ROM is typically used to store:

- Boot programs (Bootloaders);
- Processor or microcontroller initialization routines;
- Constant tables or microcodes required for internal CPU operation.

ROM is organized as a matrix of memory cells arranged in rows and columns. Each cell represents one bit (0 or 1), and its state is determined by the presence or absence of a physical connection (such as a diode or transistor).

The general operation of a ROM involves three main blocks:

- **Address decoder:** receives an *N*-bit binary address and selects the corresponding row in the matrix.
- **Memory matrix:** contains the pre-programmed cells located at the intersections of rows and columns.
- **Output multiplexer:** gathers the data read from the selected columns and sends them to the data bus.

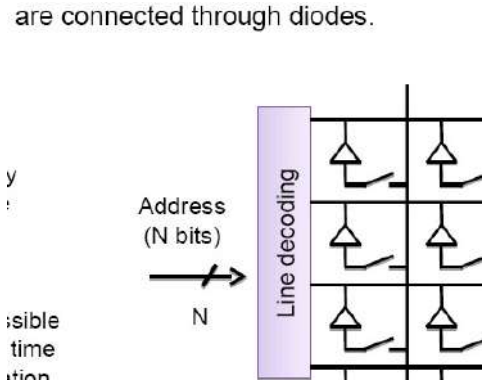


Figure VI. 5 - Simplified structure of a ROM showing address decoding and data reading

When an address is applied at the input, the decoder activates the corresponding row, allowing the multiplexer to retrieve the stored bit values (0 or 1) from the associated cells.

Thus, ROM performs only read operations, with no possibility of writing or later modification.

IV.8.1 PROM

PROM (Programmable Read-Only Memory) is an evolution of classical ROM. Unlike standard ROM, it is blank at manufacture and can be programmed once by the user using a device called a PROM programmer.

This memory type is particularly useful when the manufacturer needs to record specific programs or data after production, such as for client configurations or industrial prototypes.

PROM operates with a matrix of memory cells similar to ROM, but with electrical connections that can be permanently modified during programming. Each memory cell contains a fuse or antifuse, which is the key element used to store data, as shown in the figure below.

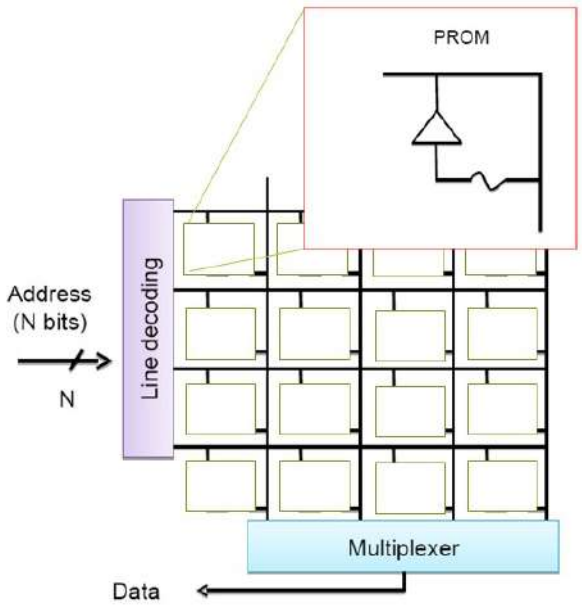


Figure VI. 6 - Simplified structure of a PROM – information encoding using fuses

In a PROM, each cell corresponds to one bit of information:

- When the fuse is intact, it represents a logical 1.

- When a high electrical pulse is applied by the programmer, the fuse blows, breaking the circuit, which corresponds to a logical 0.

This electrical burning process is irreversible: once a fuse is blown, it cannot be restored. PROMs are therefore one-time programmable (OTP) memories.

The essential components of a PROM cell are:

- The selection transistor, which identifies the cell to be programmed.
- The fuse, which stores the binary value depending on its state (intact or blown).

PROMs combine the stability of classical ROM with limited flexibility for post-manufacturing customization. However, because they cannot be reprogrammed, their use is restricted to applications where data must never change, such as industrial embedded systems, identification circuits, and fixed-configuration microcontrollers.

IV.8.2 EPROM

EPROM (Erasable Programmable Read-Only Memory) is a non-volatile memory that, unlike PROM, can be erased and reprogrammed multiple times.
It was developed to provide a flexible solution for embedded system designers, allowing them to modify the memory content without manufacturing a new chip for each update.

The key element of an EPROM is the floating-gate transistor, as illustrated in the figure below.

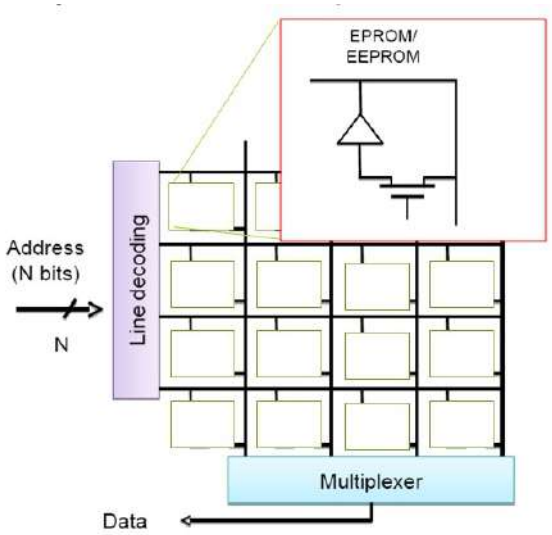


Figure VI. 7 - Simplified structure of an EPROM/EEPROM – floating-gate memory cell

Each EPROM cell is made up of a modified MOS transistor with two gates:

- a control gate, connected to the memory's control circuitry;
- a floating gate, electrically isolated by an oxide layer and capable of storing electric charges.

During programming, a high voltage is applied, causing electrons to become trapped on the floating gate. These electrons remain stored as long as no erase operation is performed, thereby modifying the transistor's conduction threshold. Depending on the presence or absence of electrons on the floating gate, the cell is interpreted as a logical 0 or 1.

The erasure of an EPROM's content is achieved using ultraviolet (UV) radiation with a wavelength of approximately 253 nm. To do this, the chip is exposed to UV light through a small quartz window integrated into its package. This radiation releases the trapped electrons in the floating gate, effectively resetting the memory for a new programming cycle.

In the figure, the EPROM cell includes:

- a selection transistor, allowing access to the cell according to the specified address;
- a floating-gate transistor, which stores the electrical charge corresponding to the data bit.

These cells are arranged in a matrix, controlled by an address decoder and connected to an output multiplexer, similar to other ROM architectures.

EPROM is therefore a non-volatile, reusable memory after complete erasure by UV exposure.
It offers excellent data stability and good charge retention, but its UV-based erasure is relatively slow (several minutes) and requires physical exposure, which limits its use in systems that need frequent updates.

Thanks to its reprogrammability and low cost, EPROM was widely used during the 1980s and 1990s for the development and prototyping of microcontrollers, before being gradually replaced by EEPROM and Flash technologies, which are faster and more convenient.

IV.8.3 EEPROM

EEPROM (Electrically Erasable Programmable Read-Only Memory) is a non-volatile memory that allows electrical erasure and reprogramming, without exposure to ultraviolet

light. It also uses floating-gate transistors capable of storing an electric charge that represents a data bit.

During programming, an appropriate voltage injects electrons into the floating gate, changing the logical state of the cell. Erasure is achieved by applying a reverse voltage that releases these electrons, restoring the cell to its initial, writable state.This operation can be performed selectively, cell by cell, without erasing the entire memory.

EEPROM offers several advantages:

- it retains data without power,
- it allows electrical updating of the content,
- it supports a large number of write and erase cycles.

However, it has slower access times and a lower storage capacity than other memory types.

EEPROMs are commonly used in embedded systems to store configuration parameters, calibration values, or persistent information that must be preserved even after a power loss.

IV.8.4 Flash Memory

Flash memory is a type of non-volatile memory derived from EEPROM technology. It retains data without electrical power and provides high storage density, fast access speeds, and quick reprogramming capabilities. Unlike EEPROM, which allows cell-by-cell erasure, Flash memory performs erasure by blocks, which reduces erase/write time and simplifies internal control.

Flash memory also relies on floating-gate transistors that store electrical charges representing binary data. During programming, a high voltage injects electrons into the floating gate, modifying the transistor's threshold voltage. Erasure is performed by applying a reverse voltage to remove the trapped electrons, restoring the cells for new programming.

Two main architectures exist depending on the organization of the memory cells: NOR and NAND.

- In NOR Flash, the cells are connected in parallel, enabling direct and random access to data. This configuration allows byte-level read and write operations,

making it ideal for executable code storage (firmware). However, it takes up more space and offers lower storage density.

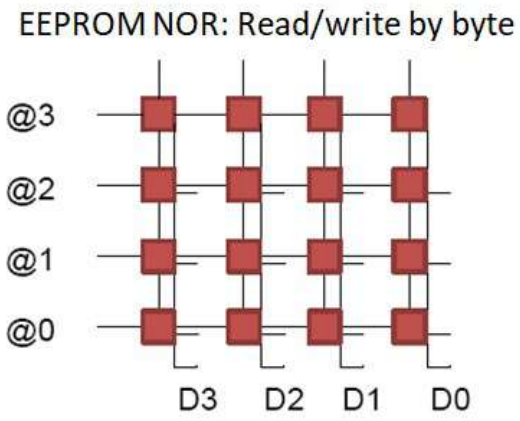


Figure VI. 8 - Organization of a NOR-type EEPROM memory – byte-level read/write operations

- In NAND Flash, the cells are connected in series within chains or blocks, allowing sector-level read/write operations. This design increases storage density and reduces manufacturing cost, while improving write speed. It is particularly well-suited for mass storage applications such as SD cards, USB drives, and SSD disks, where large amounts of data must be accessed quickly.

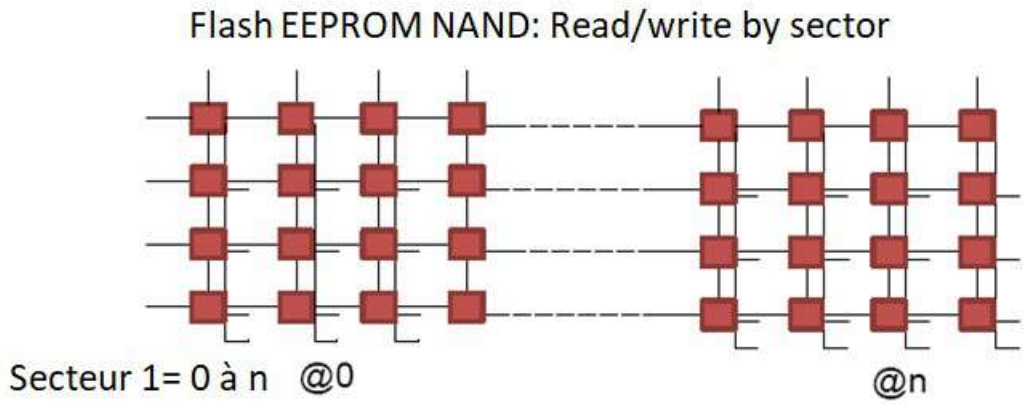


Figure VI. 9 - Organization of a NAND-type Flash memory – sector-level read/write operations

Flash memory provides numerous advantages:

- data retention without power,
- fast access,
- compact size, and

- low power consumption.

However, its lifespan is limited by the number of write/erase cycles supported by each cell, and reprogramming requires full block erasure.

In embedded systems, Flash is often integrated into microcontrollers and used to store the main program, configuration parameters, or critical data. Thanks to its stability and performance, Flash memory has become an essential component of modern embedded architectures.

IV.8.5 Mass Storage Memories

Mass storage memories are devices designed to store large amounts of data permanently, even without power. Unlike main memories such as RAM, which are used for temporary data processing, mass storage provides long-term data retention.

They are used to store files, programs, operating systems, or databases in complex embedded systems. The operation of mass memories is based on different technologies, mainly distinguished by the storage medium and access mode. The three main families are:

- Magnetic memories, such as hard disk drives (HDDs) and magnetic tapes, which use magnetic particle polarization to represent bits. These offer large capacity at low cost, but are sensitive to shocks and consume more energy, making them less suitable for modern embedded systems.
- Optical memories, such as CDs, DVDs, and Blu-ray discs, which use a laser beam to read or write microscopic patterns on the surface of the disc. They are durable and mainly used for archival storage, but less suited for active storage in embedded devices.
- Semiconductor memories, such as SSDs, SD cards, and USB drives, based on Flash technology. These have largely replaced magnetic media in most embedded applications due to their compactness, fast access, shock resistance, and low energy consumption.

In embedded systems, mass memories are essential for storing system logs, configuration data, and sensor-acquired information. Their selection depends on factors such as required capacity, access speed, reliability, and energy efficiency.

Thus, mass storage memories represent the permanent storage level of embedded systems, ensuring the preservation and availability of essential operational data.

IV.9 Volatile Memories

Volatile memories are temporary storage devices that lose their contents once power is removed. They constitute the working memory of an embedded system and play a crucial role in program execution. The processor uses them to store intermediate data, temporary variables, and currently processed instructions. Their access speed is a key factor in the system's overall performance, as they are involved in every execution cycle.

The most common type of volatile memory is RAM (Random Access Memory), which exists in several variants depending on the underlying technology and system requirements.

RAM is a *random-access* memory, meaning that each cell can be directly accessed through its address, without scanning other cells. It serves as the main memory in an embedded system and provides the workspace for the processor. Unlike non-volatile memories, it is cleared each time the system restarts.

Its organization is based on a matrix of cells arranged in rows and columns. Each cell stores a bit of information — typically using a transistor and a capacitor (in DRAM) or only transistors (in SRAM).

Two main families are distinguished:

- **DRAM (Dynamic RAM)**
- **SRAM (Static RAM)**

These differ in internal structure and performance characteristics.

IV.9.1 SRAM

SRAM (Static Random Access Memory) is a very fast volatile memory used primarily in embedded systems that require instant data access. Unlike DRAM, it does not need periodic refresh cycles to retain data, as the information remains stable as long as power is supplied.

Each SRAM cell consists of six MOS transistors forming a bistable latch capable of storing one bit. Transistors T3 and T4, combined with T5 and T6, form two cross-coupled inverters that maintain the logical state. Transistors T1 and T2 act as access gates for reading and writing operations.

When a Word Line (WL) is activated, these transistors connect the cell to the Bit Lines (BL), enabling data transfer without disturbing the stored value.

All cells are organized in a matrix, addressed through row and column decoders. The row decoder selects the word line, while the column decoder identifies the bit line. Data then passes through a multiplexer/demultiplexer (Mux/Demux) that manages data flow during read or write operations.

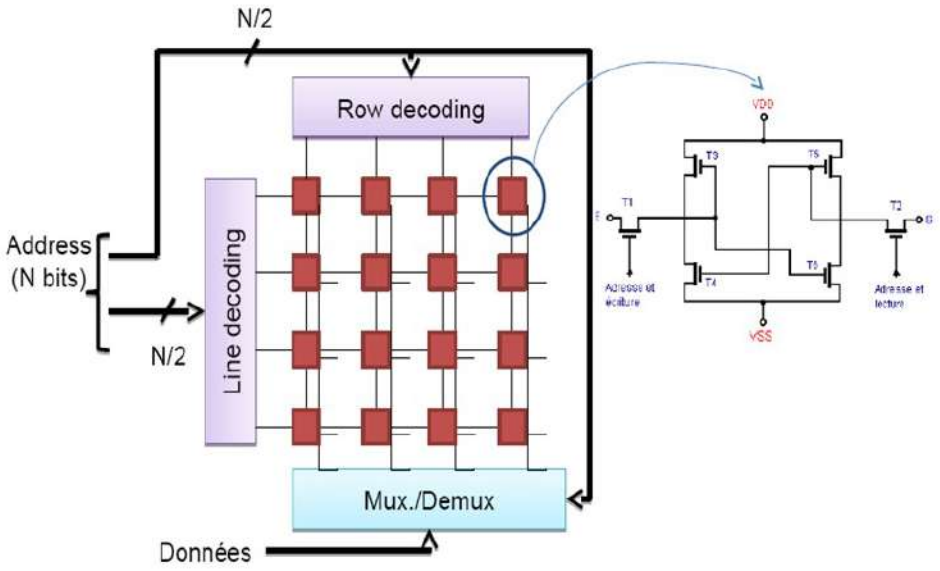


Figure VI. 10 - Internal organization of an SRAM and 6T basic memory cell

This architecture ensures high speed and data stability, at the cost of lower density and higher cost per bit compared to DRAM.

Thanks to its speed and reliability, SRAM is widely used in:

- processor cache memory (L1, L2, L3),
- internal registers,
- high-speed communication buffers, and
- embedded memories in microcontrollers, FPGAs, and ASICs.

It is thus an essential component for high-performance embedded applications.

IV.9.2 DRAM

DRAM (Dynamic Random Access Memory) is a type of volatile memory in which each cell is made up of one transistor and one capacitor. The transistor acts as a switch controlling access to the cell, while the capacitor stores the electrical charge that represents the data bit:

- a **charged capacitor** corresponds to a logical **1**,
- a **discharged capacitor** corresponds to a logical **0**.

Because the capacitor's charge gradually leaks away, the contents of DRAM must be refreshed periodically to avoid data loss. This refresh operation makes DRAM slower than SRAM, but allows greater density and lower cost per bit.

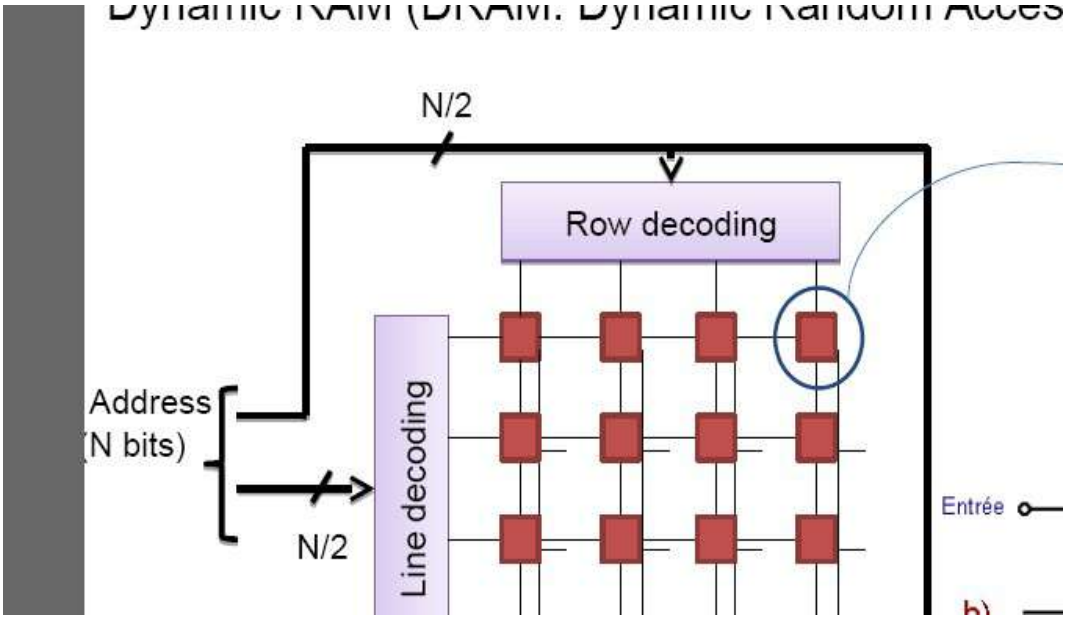


Figure VI. 11 - Structure of a DRAM and its basic 1T/1C memory cell

The DRAM matrix is composed of numerous cells arranged in rows and columns, which are selected by decoding circuits. A multiplexer/demultiplexer manages read and write operations, while the cell structure on the right shows the dynamic storage principle.

This architecture makes DRAM the preferred choice for main memory in many embedded systems requiring large capacity at low cost.

There are two primary types of DRAM, distinguished by their operational mode:

- Asynchronous DRAM, where read and write operations are not synchronized with a clock signal. The processor must wait for one cycle to complete before starting another.
- Synchronous DRAM (SDRAM), which operates in sync with the processor clock, enabling fast and continuous data access.

Because of its simple yet efficient design, DRAM remains ubiquitous in embedded systems, offering a balance between capacity, cost, and compatibility with a wide range of processors and microcontrollers.

A) Asynchronous DRAM

Asynchronous DRAM is the oldest and simplest form of dynamic memory. Its name comes from the fact that its operation is not synchronized with any external clock. Read and write operations are controlled by two main signals:

- **RAS (Row Address Strobe)**: activates the target row,
- **CAS (Column Address Strobe)**: activates the target column.

In an asynchronous DRAM, the processor must wait for each access cycle (read or write) to complete before initiating the next one. This results in longer access times, since each operation must be fully processed before moving to the next.

However, the simplicity and low cost of this architecture made it popular in early embedded systems and in applications where speed was not critical. Notable variants include:

- FPM DRAM (Fast Page Mode) – improves performance by keeping a row active for multiple consecutive accesses.
- EDO DRAM (Extended Data Out) – further reduces cycle time by preparing the next data access while the current one is still being read.

These improvements paved the way for Synchronous DRAM (SDRAM), which operates in coordination with the system clock, providing significantly better performance while maintaining the same basic structure.

B) Synchronous DRAM (SDRAM)

SDRAM (Synchronous DRAM) is a major improvement over asynchronous DRAM, as it operates synchronously with the processor clock. This synchronization allows ordered read/write operations, eliminating idle waiting periods and enabling seamless data transfers. Multiple accesses can be chained together without interruption, significantly increasing transfer speed and memory bandwidth.

Structurally, SDRAM retains the dynamic memory principle — a cell made of one transistor and one capacitor — but adds clock-synchronized control circuits.

These circuits manage data flow efficiently through pipelining mechanisms, which prepare one access while the previous one is still being executed.

Successive generations of SDRAM have brought major improvements:

- **DDR (Double Data Rate SDRAM)** – doubles throughput by transferring data on both the rising and falling edges of the clock.
- **DDR2, DDR3, DDR4, and DDR5** – progressively increase frequency, reduce supply voltage, optimize power efficiency, and expand storage density.

Thanks to these developments, SDRAM and its derivatives have become the standard memory in modern embedded and computing systems. They are widely used in development boards (e.g., Raspberry Pi, high-end STM32), embedded processors, and real-time data processing systems such as robotics, computer vision, and communication equipment.

IV.10 Common Memory Issues in Embedded Systems

Embedded systems can face several critical problems related to memory management.

One major issue is memory fragmentation, which occurs when free space becomes divided into small, non-contiguous blocks, preventing the allocation of new memory areas even if total capacity seems sufficient. This often happens in systems using dynamic memory allocation.

Another frequent problem is memory leakage, which occurs when allocated memory blocks are not released after use. Over time, this reduces available memory, leading to slower performance or system crashes. To prevent these issues, good management practices are essential:

- minimize dynamic allocation,
- favor static data structures,
- monitor pointer integrity, and
- use memory debugging tools to detect errors.

In critical systems, strict memory management is indispensable to ensure long-term reliability and stability.

IV.11 Advanced Aspects of Memory Management

Modern embedded architectures integrate advanced mechanisms to optimize memory use and efficiency.

Virtual memory allows simulation of a capacity greater than physical memory by using secondary storage, enhancing flexibility and multitasking capabilities. Prefetching techniques anticipate processor needs by preloading data likely to be requested soon, thereby reducing wait times during execution.

Memory access optimization also involves:

- instruction and data caching,
- minimizing access conflicts, and
- hierarchical memory organization (L1, L2, L3).

These methods improve system responsiveness while reducing power consumption. In demanding embedded applications — such as automotive systems, robotics, and real-time processing — mastering these memory management techniques is key to achieving high performance and reliability.

Chapter V: Communication Buses

V Communication Buses

In an embedded system, the various components — processor, memory, and peripherals — must exchange information smoothly and in a coordinated manner. This exchange relies on a set of electrical lines called communication buses, which act as data transport networks within the system.

A bus is therefore a mechanism for transferring information between the different elements of a system. It generally consists of multiple wires (or PCB traces) carrying address, data, and control signals. Each bus is characterized by its width (number of lines), the type of information it transmits, and its mode of operation — parallel or serial, synchronous or asynchronous.

Choosing the appropriate bus is essential, as it determines the speed of communication, energy consumption, and system stability. In modern embedded architectures, the communication bus directly influences the overall performance of the device.

Communication between a processor and a peripheral is based on the coordinated exchange of three main types of signals: address, data, and control. These signals circulate through the system's buses, as illustrated below.

The processor first sends an **address** to identify the resource or peripheral involved. This address locates the register or memory zone where the operation will take place. Depending on the type of operation, the processor may then:

- **Write data** (transfer from processor to peripheral), or
- **Read data** (transfer from peripheral to processor).

The R/W (Read/Write) control line indicates the direction of the transfer:

- **R (Read) → data are read from the peripheral.**
- **W (Write) → data are written to the peripheral.**

This tripartite organization — **address bus**, **data bus**, and **control bus** — ensures clear, reliable, and synchronized communication among system components. It forms the foundation of all interactions between the microprocessor, memory, and I/O interfaces, both in simple architectures and in complex embedded systems.

V.1 Communication Principle – Read Cycle

When the processor needs to read data from memory or a peripheral, the operation follows a precise temporal sequence coordinated by the address, data, and control signals.

1. The processor first places a valid address on the address bus to indicate the data location.
2. The $R\overline{D}$ (Read) control signal is then asserted low, indicating that a read operation is underway. This signal enables the addressed memory or peripheral to place the requested data on the data bus.
3. Once the data become valid, the processor reads it before the $R\overline{D}$ signal returns high. When $R\overline{D}$ is deasserted, the peripheral releases the data bus, completing the read cycle.

All these steps are synchronized with the system clock, which defines the timing intervals of the transfer — known as machine cycles (T1, T2, T3, etc.).

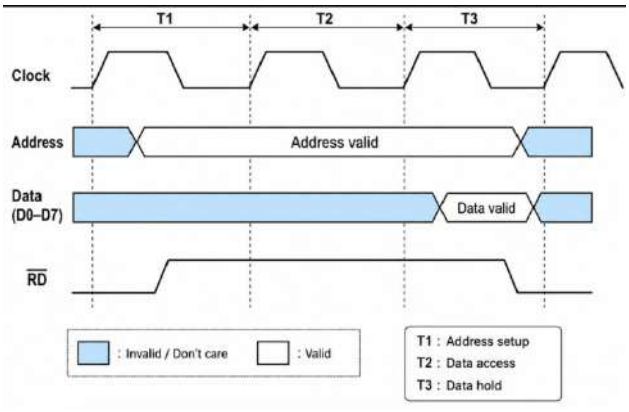


Figure V. 1 - Read cycle timing diagram

During T1, the processor outputs the address.

During T2, the data become available on the bus.

During T3, the processor reads the data before the bus is released.

This sequence ensures reliable, orderly, and synchronized data transfer within the embedded system.

V.2 Communication Principle – Write Cycle

The write cycle corresponds to the process by which the processor sends data to a memory or peripheral for storage.

4. The processor outputs a **valid address** on the address bus to identify the target register or memory cell.
5. The data to be written are placed on the data bus.
6. The **R/$\overline{W}$** control signal is driven low to indicate a write operation. The data must remain stable during the active period of this signal to ensure proper storage.
7. Once the operation is completed, **R/$\overline{W}$** returns high, marking the end of the cycle.

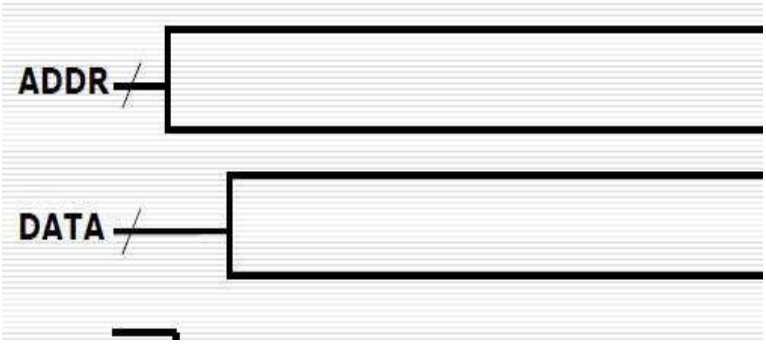


Figure V. 2 - Write cycle timing diagram

This communication process is fundamental in embedded systems, as it allows the processor to update registers, save variables, and configure external peripherals. A precise understanding of the read and write cycles is essential for ensuring both reliability and performance of the overall system.

V.3 Types of Communication Buses

Buses are the essential communication pathways between the various components of an embedded system. They ensure the orderly transfer of data, addresses, and control signals between the processor, memory, and peripherals.

There are two main categories of buses:

- Internal buses, used within the microprocessor or microcontroller;
- External buses, which connect the processor to peripheral devices and external memories.

V.3.1 Internal Bus

The internal bus connects the functional units of the processor: the Arithmetic and Logic Unit (ALU), registers, cache memory, and the control unit.
It enables very high-speed exchanges, ensuring consistency between the instructions and data processed by the CPU.

One of the most common internal buses in modern ARM architectures is the AHB (Advanced High-performance Bus), defined in the AMBA standard (Advanced Microcontroller Bus Architecture).

A) The AHB Bus (Advanced High-performance Bus)

The AHB bus is designed to provide fast and efficient transfers between the processor core and internal components such as memory and high-performance peripherals.
It serves as the backbone of communication inside the microcontroller.

Main characteristics of the AHB bus:

- High-performance transfers: supports high data rates through a *pipelined* operation, allowing a new transaction to begin before the previous one finishes.
- Variable width: typically 32 or 64 bits, enabling multiple bytes to be transferred per clock cycle.
- Centralized arbitration: an AHB controller grants the bus to only one master (CPU, DMA, etc.) at a time, preventing conflicts.
- Burst mode: allows sequential block transfers to increase bandwidth.
- Unified interface: all connected peripherals follow a single standardized protocol, simplifying integration.

Typical applications in embedded systems:

- Communication between an ARM Cortex-M processor and the internal Flash memory for code execution.
- Direct data transfer between SRAM and a DMA controller, reducing CPU workload.

- Data exchange between a graphic controller and video memory in embedded systems with displays.

Thus, the AHB bus ensures fast and stable internal communication, which is essential for real-time embedded systems.

V.3.2 External Bus

The external bus enables communication between the microcontroller and external peripherals such as sensors, actuators, external memories, or communication modules (Wi-Fi, GPS, Bluetooth, etc.). It can operate in parallel mode (multiple bits transmitted simultaneously) or serial mode (data transmitted bit by bit). In modern embedded systems, serial buses are preferred for their simplicity and low power consumption.

V.3.3 SPI Bus (Serial Peripheral Interface)

The SPI bus is a synchronous serial communication protocol widely used to connect a microcontroller to high-speed external peripherals. It operates in a master–slave configuration, where the microcontroller (master) manages communication with one or more peripherals (slaves).

Main lines:

- **MOSI (Master Out Slave In):** data sent from the master to the slave.
- **MISO (Master In Slave Out):** data sent from the slave to the master.
- **SCK (Serial Clock):** clock signal generated by the master to synchronize communication.
- **SS (Slave Select):** selection line that activates the target peripheral.

Advantages of the SPI bus:

- Very high data rate (tens of MHz).
- Simple interface to implement and low power consumption.
- Ideal for short-distance communication, making it perfect for compact embedded systems.

Typical applications:

- Communication between a microcontroller and an SPI Flash memory for data storage.

- Connection of an inertial sensor (IMU) or accelerometer in mobile robots.
- Interface with TFT or OLED displays in embedded visualization systems.

V.3.4 Serial Bus

A serial bus transmits data bit by bit over one or more lines, synchronized by a clock or a defined timing protocol. This transmission mode significantly reduces the number of wires, thereby lowering cost, size, and electromagnetic interference.

In a serial communication, binary information is transmitted sequentially on the same line, with synchronization ensured either by a shared clock or timing rules. This makes serial buses particularly suitable for compact and reliable embedded systems.

They are used to connect the microcontroller to external peripherals such as sensors, memories, displays, or communication modules.

A serial bus can be:

- Synchronous, such as SPI (Serial Peripheral Interface) or I²C (Inter-Integrated Circuit); or
- Asynchronous, such as UART (Universal Asynchronous Receiver-Transmitter).

Examples of serial buses in embedded systems:

- SPI is widely used to connect a microcontroller to an external Flash memory, TFT display, or analog-to-digital converter.
- I²C connects multiple sensors (temperature, pressure, humidity) using only two lines (SDA and SCL), reducing pin usage.
- CAN (Controller Area Network) is common in automotive systems, ensuring reliable communication between ECUs with high noise immunity.
- UART interfaces are used for communication with GPS, GSM, or Bluetooth modules.

Due to its reliability, compactness, and low wiring complexity, the serial bus is the preferred solution for external communication in most modern embedded systems, especially when the required transfer rate remains below a few hundred megahertz.

V.3.5 Parallel Bus

A parallel bus allows simultaneous transfer of multiple bits across several distinct lines. Each bit is transmitted on a separate wire, enabling high data throughput per clock cycle. This architecture makes the parallel bus ideal for high-speed internal communication between the processor and memory, as well as certain external interfaces.

A parallel bus typically includes data, address, and control lines.
At each clock pulse, all bits of a word are transmitted simultaneously, ensuring fast and efficient communication. However, this method requires a large number of physical connections, which increases wiring complexity and susceptibility to noise.
At higher frequencies, synchronization among all lines becomes increasingly difficult.

In embedded systems, parallel buses are commonly used to connect:

- Microcontrollers to external SRAM or Flash memory, or
- Microcontrollers to parallel LCD displays.

For example, some ARM-based embedded boards use a 16- or 32-bit external parallel bus to interface with external memory, extending system storage capacity.
This configuration is common in data loggers, industrial display panels, and embedded graphics controllers.

In systems with graphical displays, the parallel LCD bus (FSMC – *Flexible Static Memory Controller*) is often used to transfer images or display data at high speed between the microcontroller and the screen module. This bus supports 8-, 16-, or 32-bit transfers, ensuring high display responsiveness, which is crucial for Human–Machine Interfaces (HMI).

While parallel buses offer very high throughput, they come with higher power consumption and routing complexity. In modern embedded architectures, they are often replaced or complemented by high-speed serial buses such as SPI or QSPI, which reduce wiring while maintaining satisfactory transfer rates.

V.4 Bus Characteristics

Communication buses form the foundation of data exchange in embedded systems. Their performance mainly depends on three key parameters: bus width, transfer rate, and communication protocol. These characteristics directly affect the speed, reliability, and energy efficiency of the overall system.

V.4.1 Bus Width

The bus width corresponds to the number of physical lines used to transmit data simultaneously.
The greater the width, the larger the amount of data transferred per clock cycle. For example, an 8-bit bus transfers one byte per cycle, while a 32-bit bus transfers four bytes.

In modern microcontrollers, such as those based on ARM Cortex-M cores, the internal bus width is typically 32 bits, offering a good balance between speed and power consumption. However, increasing the number of lines makes the wiring more complex and raises energy consumption.

Therefore, low-power systems, such as autonomous sensors or IoT devices, tend to use narrower buses (8 or 16 bits), whereas high-performance systems, such as industrial control processors, adopt 64-bit buses to maximize bandwidth.

V.4.2 Transfer Rate

The transfer rate represents the amount of data that can be exchanged per unit of time, expressed in bits per second (bps). It depends on the clock frequency, bus width, and the nature of communication (serial or parallel).

In an embedded system, this rate determines memory access speed, communication fluidity with peripherals, and the overall responsiveness of the device.

For instance:

- An SPI bus can reach tens of megahertz,
- Whereas an I²C bus is often limited to a few hundred kilohertz.
- Internal buses such as AHB or AXI, used in ARM architectures, can achieve hundreds of megahertz, which is essential for fast data processing.

An appropriate transfer rate ensures effective synchronization between modules and minimizes communication delays.

V.4.3 Communication Protocols

Communication protocols define the rules governing exchanges between components connected to a bus. They specify how data are encoded, addressed, synchronized, and

validated to ensure reliable transmission. Each bus employs a specific protocol suited to its functional requirements.

For example:

- The SPI (Serial Peripheral Interface) protocol enables high-speed communication between a microcontroller and external devices such as memories or displays.
- The I²C (Inter-Integrated Circuit) protocol, using only two wires, is widely employed to connect multiple sensors on the same bus, each identified by a distinct address.
- The CAN (Controller Area Network) protocol is robust and error-tolerant, making it ideal for critical automotive communications.
- Finally, internal buses standardized by ARM, such as AHB and AXI, rely on optimized protocols that guarantee fast and orderly data transfers between the processor, memory, and peripherals.

V.5 Communication Protocols

V.5.1 UART (Universal Asynchronous Receiver-Transmitter)

UART is a widely used asynchronous serial communication protocol in embedded systems for data exchange between a microcontroller and a peripheral (computer, Bluetooth module, GPS, sensor, etc.). Unlike synchronous protocols such as SPI or I²C, UART does not require any shared clock signal — bit synchronization is achieved through start and stop signals transmitted within each frame.

UART communication is based on two main lines:

- **TX (Transmit):** line used to send data.
- **RX (Receive):** line used to receive data.

A common ground (GND) connects both devices to ensure a shared electrical reference.

Communication is generally point-to-point, meaning between only two elements: a transmitter and a receiver. Each bit is transmitted sequentially on the TX line, while the RX line of the receiving device captures the data.

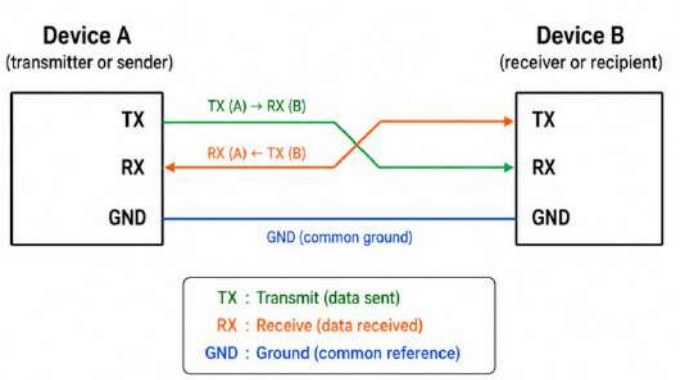


Figure V. 3 - UART connection diagram between two elements

This figure illustrates the typical **crossed connection** between two UART devices:

- The TX output of element A (for example, a microcontroller) is connected to the RX input of element B (such as a GPS module).
- Conversely, the TX output of element B is connected to the RX input of element A.

This crossed configuration enables bidirectional communication between the two devices.

A UART communication frame consists of several successive parts allowing the reliable transmission of one byte of data:

- 1 start bit: indicates the beginning of transmission (transition from high to low).
- 5 to 9 data bits: contain the useful information, sent from the least significant bit (LSB) to the most significant bit (MSB).
- 1 parity bit (optional): used for simple error detection.
- 1 or 2 stop bits: indicate the end of transmission (return to the high level).

During idle periods, the TX line remains at a logical high state.

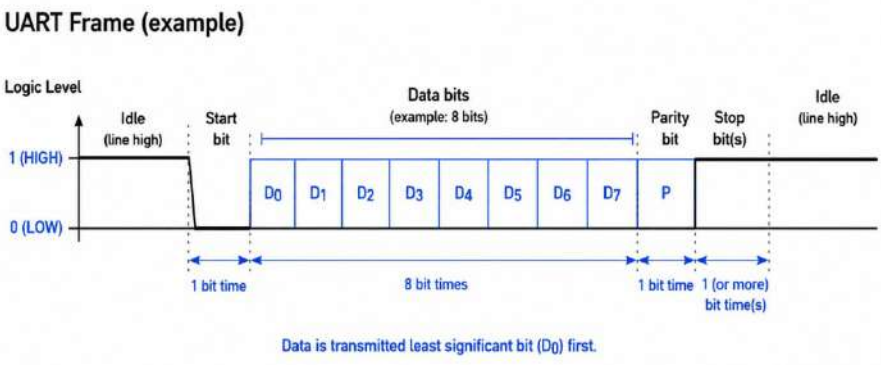


Figure V. 4 - Frame of a UART serial transmission (sequence)

This figure illustrates the typical timing diagram of a UART frame. It shows the succession of the different bits composing a frame:

- The start bit drives the line low, triggering message reception.
- Data bits (D0 to Dn) are then transmitted one after another.
- The parity bit, when enabled, allows simple error detection during transmission.
- Finally, the stop bit returns the line to the high state, marking the end of the message.

Transmission speeds, expressed in baud rates, vary depending on the system — from 1200 to 115200 baud (and even higher in modern systems). Both devices must be configured with identical parameters (speed, number of bits, parity, etc.) to ensure reliable communication.

V.5.2 SPI (Serial Peripheral Interface)

The SPI (Serial Peripheral Interface) is a synchronous serial communication protocol widely used in embedded systems to connect a master microcontroller to multiple slave devices, such as sensors, memories, displays, or analog-to-digital converters.

It is based on a master-slave architecture, where the master controls synchronization and peripheral selection via a shared clock and control lines.

Main SPI bus lines:

- SCLK (Serial Clock): clock generated by the master to synchronize communication.
- MOSI (Master Out Slave In): line used to send data from the master to the slave.
- MISO (Master In Slave Out): line used by the slave to send data to the master.
- SS (Slave Select): signal activated by the master (low level) to select the desired slave.

When the SS line is set low, the corresponding slave becomes active, while the other peripherals remain inactive. Data transfer occurs bit by bit with each clock pulse, making communication fast, reliable, and full-duplex (simultaneous transmission and reception).

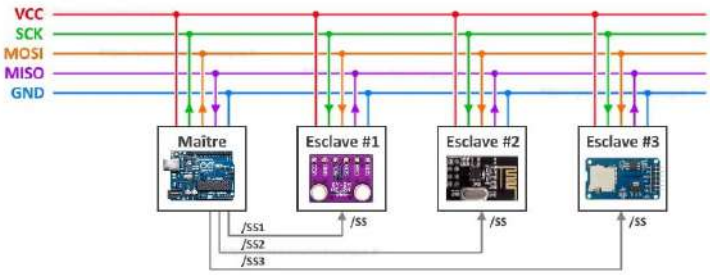


Figure V. 5 - SPI communication architecture between master and slave

This figure illustrates the fundamental structure of the SPI bus connecting a master to several slaves through the four communication lines. The master generates the clock (SCK) and controls the MOSI and MISO lines, while each slave has an independent SS line. This configuration allows selective communication: only one slave is active at a time, thus avoiding data collisions.

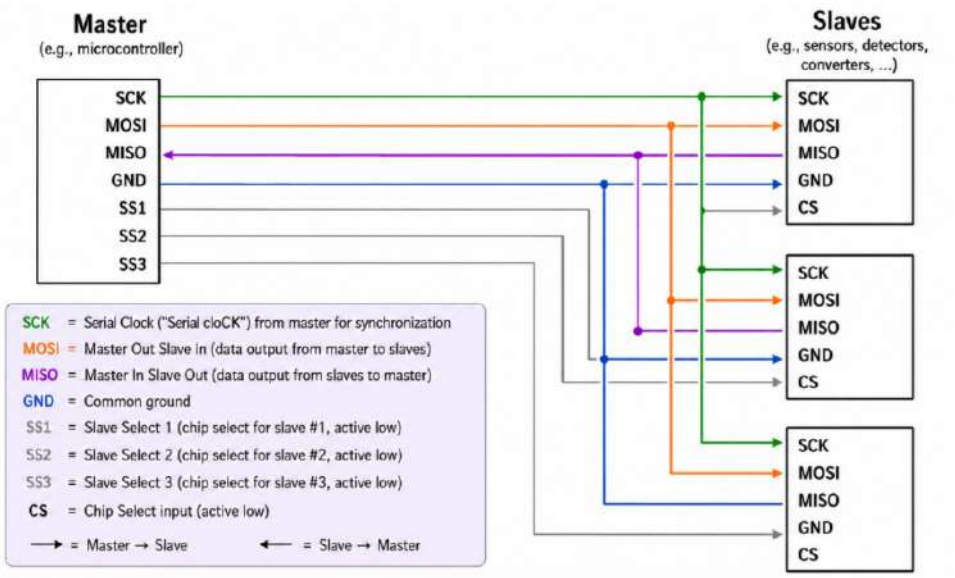


Figure V. 6 - Example of SPI wirng (with 1 master and 3 slaves)

This figure shows the practical implementation of the SPI bus with a master microcontroller and three slave peripherals. The lines SCK, MOSI, MISO, and GND are common to all peripherals, simplifying wiring. However, each slave has a dedicated SS line (/SS1, /SS2, /SS3).

When communication is established, the master pulls down the SS line corresponding to the desired slave, while the other SS lines remain high, deactivating unrelated slaves. This mechanism allows precise control and prevents interference between peripherals, at the cost of having one SS line per slave.

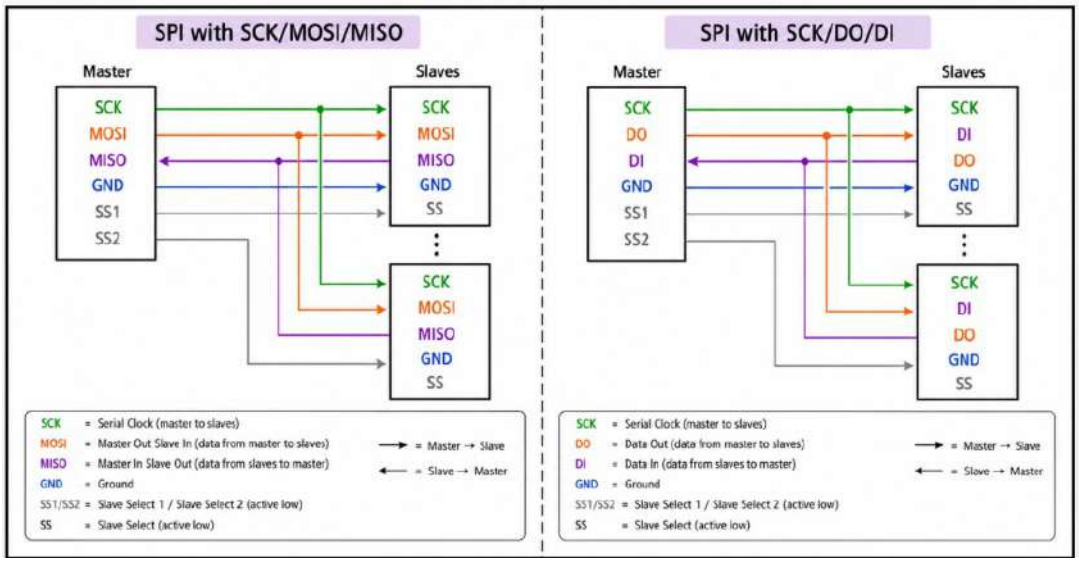


Figure V. 7 - Different SPI input/output notations

This figure compares two naming conventions used by different manufacturers:

- SCK/MOSI/MISO (the most common notation),
- SCK/DO/DI (an alternative but equivalent notation).

Thus:

- MOSI = DO (Data Out) corresponds to the master's data output.
- MISO = DI (Data In) corresponds to the master's data input.
- SCK remains the common clock line for all implementations.
- SS (or CS) designates the slave select line.

This figure highlights the compatibility of terminologies used by various manufacturers (Microchip, Atmel, STMicroelectronics, etc.), showing that despite naming differences, the logical structure and operating principles remain identical.

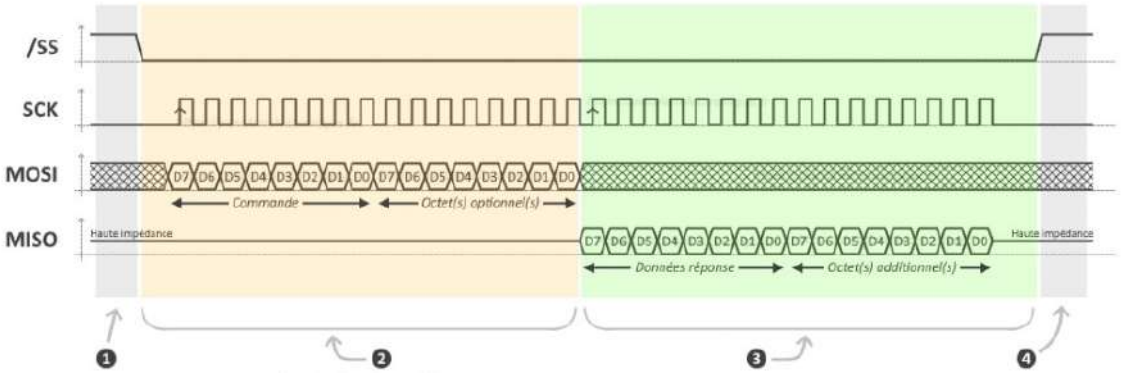


Figure V. 8 - Example of reading on the SPI bus

This figure illustrates the timing sequence of an SPI read operation:

- Initialization: the master pulls the SS line low to select a specific slave.
- Command transmission: the master sends an instruction via MOSI (e.g., a request to read a memory register).
- Reception: after receiving the command, the slave sends the requested data on the MISO line, bit by bit, synchronized with SCK pulses.
- End of communication: when the read operation is complete, the master sets the SS line high to deactivate the slave.

This figure clearly illustrates the synchronous and bidirectional nature of SPI: each data bit is transferred in phase with the clock, ensuring fast and error-free synchronization.

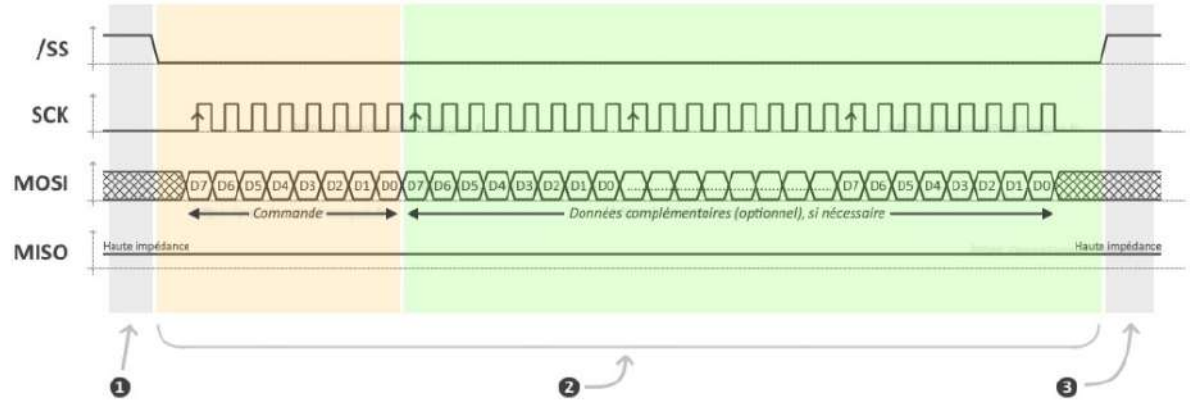


Figure V. 9 - Example of writing on the SPI bus

The SPI write operation is similar to reading, but this time the master sends data to the slave:

- The master first pulls the SS line low, thus selecting the target slave.
- It then sends one or more data packets (typically 8 bits each) on MOSI, possibly accompanied by additional parameters.
- Each bit is sent with every clock pulse on the SCK line.
- At the end of the transmission, the master sets the SS line high, indicating the end of communication.

This figure also shows that the slave maintains the MISO line in high impedance during writing, so as not to interfere with the signal transmitted on MOSI.

This sequential and strictly clocked operation ensures reliable exchanges, even at speeds reaching several tens of megahertz.

Here is the complete and precise English translation of your section on I²C (Inter-Integrated Circuit) — preserving all technical meaning, structure, and formatting exactly as in your French text:

V.5.3 I²C (Inter-Integrated Circuit)

The I²C (Inter-Integrated Circuit) protocol, developed by Philips (now NXP), is a synchronous serial communication interface widely used in embedded systems to interconnect multiple integrated circuits on the same board. It is characterized by its simple wiring and its ability to manage multiple slave devices using only two lines:

- **SDA (Serial Data):** bidirectional data transfer line.
- **SCL (Serial Clock):** clock line generated by the master to synchronize communication.

Each connected peripheral has a unique address, allowing the master to communicate individually with it. The I²C bus is therefore multi-master / multi-slave, although in practice only one master is active at a time.

Operating Principle

I²C communication is based on a master–slave exchange:

- The master initiates transactions (by sending a START and STOP bit).
- The slaves respond according to their assigned address.
- Data are exchanged bit by bit on the SDA line, synchronized by the SCL clock. Each byte sent is followed by an acknowledgment bit (ACK/NACK) confirming receipt.

Figure V. 10 - Alternative representation of the I²C bus

This figure illustrates a typical I²C bus topology connecting a master microcontroller (e.g., Arduino) to several slave devices: a temperature sensor, an OLED display, and a digital-to-analog converter (DAC). The SDA and SCL lines are shared by all modules, while the Rsda and Rscl resistors ensure logical pull-up to VCC. These resistors are essential to maintain correct logic levels because I²C peripheral outputs use open-drain configurations.

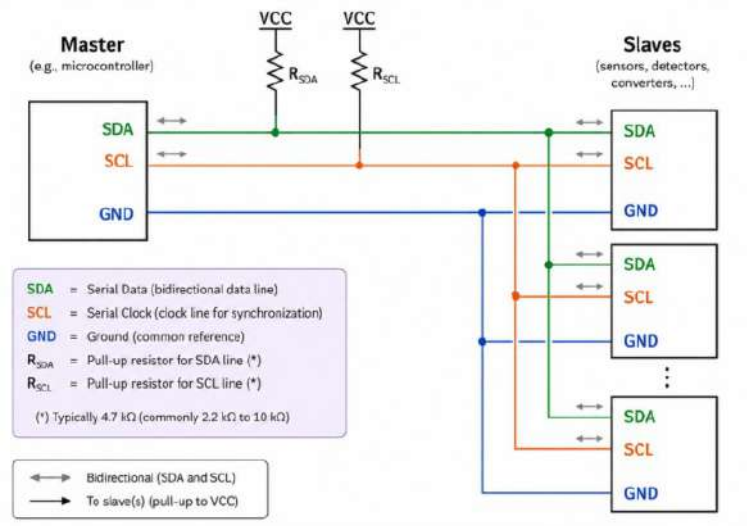


Figure V. 11 - Example of wiring for a multi-point I²C link

This figure shows the complete electrical structure of the bus: the SDA and SCL lines are common to all devices, and two pull-up resistors (typically 4.7 kΩ) are connected to the VCC power supply. The master (microcontroller) and the slaves (sensors or modules) share the same GND, ensuring proper signal synchronization. This schematic illustrates both the bidirectional nature of the SDA bus and the centralized synchronization via SCL.

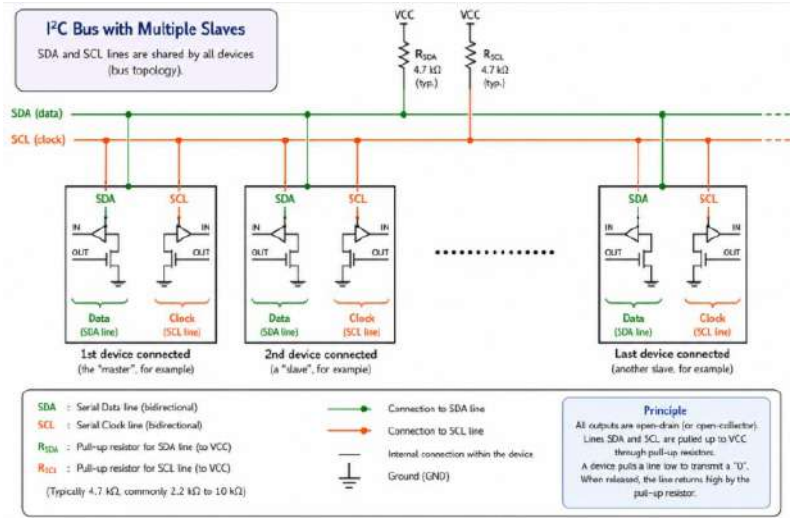


Figure V. 12 - Open-drain outputs on SDA and SCL lines

This diagram highlights how open-drain outputs operate on I²C lines. Each connected element (master or slave) includes a transistor output that can only pull the line low. The high level is achieved through the pull-up resistors (Rsda and Rscl). This mechanism allows multiple devices to share the bus safely without short-circuiting, since none actively drives a high level — the lines remain "free" when no device pulls them down. This principle enables multi-master operation and bus arbitration on I²C networks.

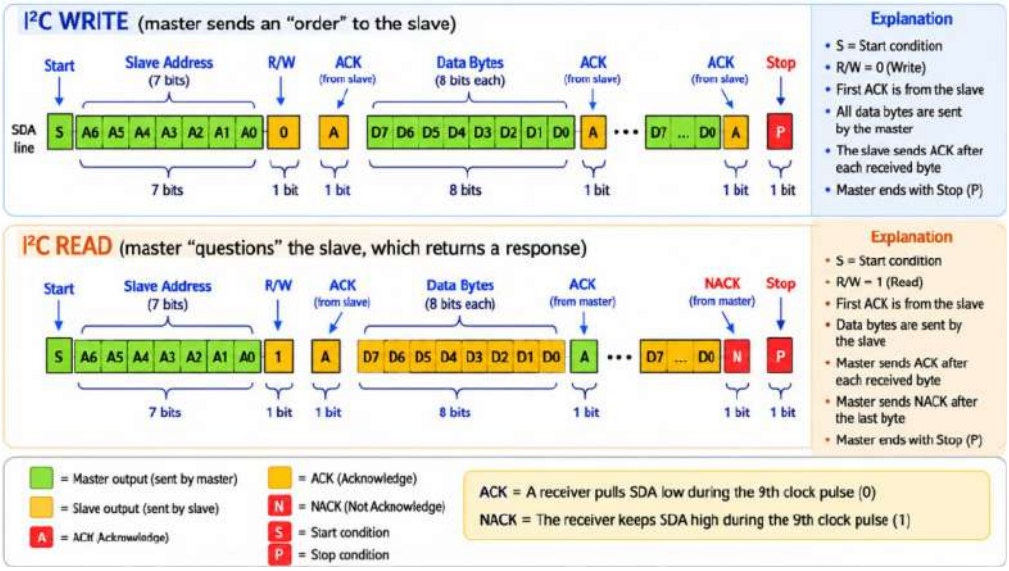


Figure V. 13 - Reading and writing on the I²C bus

This figure explains the two main phases of the protocol:

- Write Operation: The master sends a command to the slave, composed of its 7-bit address, a R/W bit (set to 0 for write), followed by the data to be transferred. The slave confirms receipt with an ACK bit.
- Read Operation: The master queries a slave (R/W bit = 1) and then reads the data transmitted by the slave.

Chapter VI: Input /Output

VI Input/Output

In an embedded system, input/output (I/O) interfaces are essential components that enable the microcontroller to interact with its environment. They provide the link between the physical world (sensors, actuators) and the digital world (processing, computation, control). Each I/O port transmits, converts, or controls information depending on the nature of the signal — analog, digital, serial, or parallel.

The proper functioning of an embedded system heavily depends on the quality, speed, and reliability of its I/O interfaces, especially in critical applications such as automotive, robotics, aerospace, and medical systems.

The figure below schematically illustrates the functional principle of input/output in a microcontroller-based system. Two types of ports can be distinguished:

- **Input ports**, connected to input peripherals (such as a push button or a sensor), allow the microcontroller to receive information from the external world.
- **Output ports**, connected to output peripherals (such as a display or a motor), allow the microcontroller to transmit control signals to the physical environment.

Thus, the microcontroller acts as a central decision unit that receives data through its inputs, processes it according to the program stored in memory, and then sends the corresponding results or control actions through its outputs. This architecture, typical of embedded systems, forms the basis of all human–machine or machine–machine interactions.

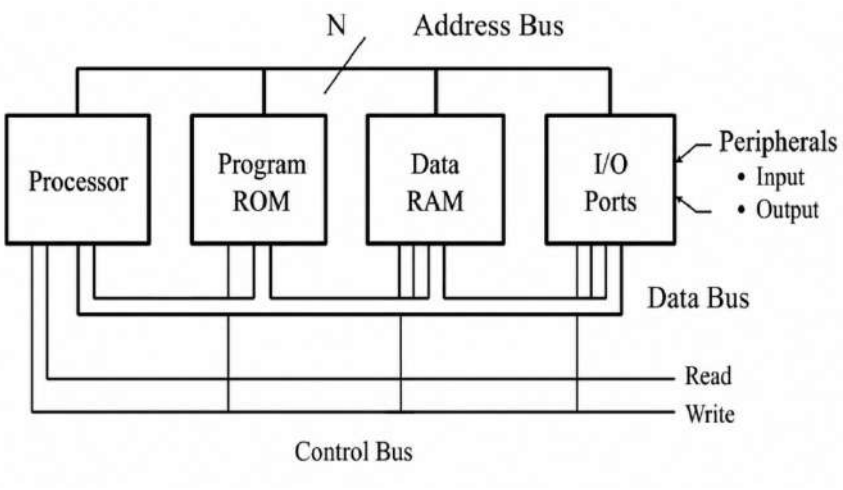


Figure VI. 1 - Functional diagram of input/output in a microcontroller-based system

VI.1 Serial and Parallel Transmission

In embedded electronics, data transmission between a microcontroller and its peripherals can be performed in two main ways: parallel or serial.

In parallel transmission, multiple bits of data (commonly 8, 16, or 32) are sent simultaneously, each on its own dedicated line. For example, transmitting one byte (8 bits) requires eight data lines, usually accompanied by several control lines for read/write, clock, or enable signals. This method offers a very high transfer rate, as several bits are transmitted per clock cycle. It is therefore preferred for fast, short-distance exchanges, such as those between the processor and internal memory within a microcontroller.

However, it also presents limitations: it requires a large number of physical connections, making wiring more complex and increasing the size of the printed circuit board (PCB). Moreover, maintaining synchronization between lines becomes challenging over longer distances due to electromagnetic noise. Parallel transmission is mainly found in internal buses of microcontrollers (e.g., AHB, APB, or AXI in ARM architectures), in processor-memory communication (RAM, ROM, EEPROM), or in high-speed interfaces for peripherals such as analog-to-digital converters (ADC) and LCD displays.

In contrast, serial transmission sends data bit by bit over one or two lines, significantly reducing the number of pins required and simplifying hardware design. Although generally slower than parallel transmission, it offers better noise immunity and allows reliable communication over longer distances. This approach is now widely used in

modern embedded systems for communication between boards, sensors, or external modules. Protocols such as SPI, I²C, UART, or CAN are based on this principle, each with specific characteristics depending on speed, distance, and device type.

Serial transmission can be:

- **Synchronous**, when a shared clock (as in SPI or I²C) provides synchronization; or
- **Asynchronous**, when it relies on start and stop bits in each frame (as in UART).

In practice, serial links are used to connect a microcontroller to digital sensors (temperature, pressure, luminosity), to communicate with external memories or displays, or to interface communication modules such as Wi-Fi, Bluetooth, or GPS. In industrial and automotive applications, the CAN bus is a typical example of a differential serial transmission, ensuring robustness and reliability in electrically noisy environments.

Thus, parallel transmission is preferred for internal high-speed communications requiring wide bandwidth, whereas serial transmission is ideal for external interconnections, where simplicity, robustness, and flexibility are more important than raw speed.

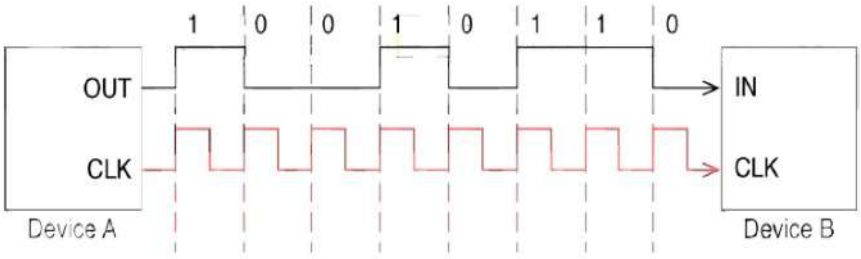
To better visualize their fundamental differences, the following table provides a comparative summary between serial and parallel transmission:

VI.2 Synchronous and Asynchronous Interfaces

I/O interfaces establish a data exchange between two entities: a transmitter (typically the microcontroller) and a receiver (sensor, memory, communication module, etc.). The key difference between the two types of interfaces lies in the presence or absence of a shared clock signal for synchronization.

VI.2.1 Synchronous Interfaces

A synchronous interface uses a common clock signal to coordinate data exchange between the transmitter and the receiver. The clock, usually generated by the microcontroller (master), defines the precise moment when each bit must be transmitted or read.
As a result, both devices operate in perfect coordination, ensuring fast, stable, and reliable communication. This type of interface is widely used in embedded systems that require continuous and high-speed data exchange.



Main advantages include:

- Precise synchronization between communicating devices.
- High transfer speed, suitable for high-frequency sensors and fast memories.
- Reduced risk of synchronization errors.

However, the presence of an additional clock line and the need for a master device make implementation slightly more complex.

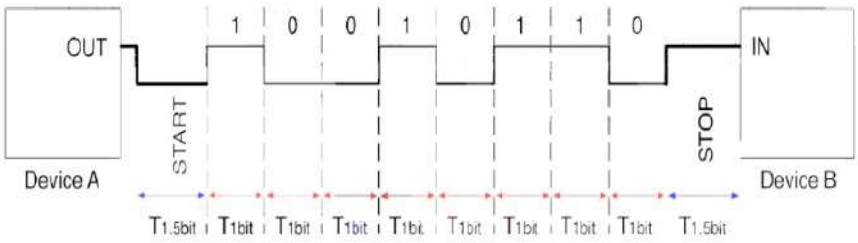
Common synchronous protocols in embedded systems include:

- SPI (Serial Peripheral Interface): a fast, full-duplex protocol commonly used for Flash memories and high-performance sensors.
- I²C (Inter-Integrated Circuit): a half-duplex, multipoint protocol widely used for communication among multiple peripherals on the same bus.

VI.2.2 Asynchronous Interfaces

An asynchronous interface, on the other hand, does not rely on a shared clock signal. Each device manages its own transmission and reception timing, which simplifies wiring but requires software-based synchronization mechanisms. Data is transmitted in the form of frames, containing start and stop bits, which allow the receiver to detect the beginning and end of each binary word.

This communication mode is particularly suited to simple, low-cost, and medium-speed links, typically used for exchanges between a microcontroller and a computer, or between two independent modules.



Its advantages include:

- Simple wiring, since no clock line is required.
- Greater flexibility for devices operating at different speeds.
- Cost-effective implementation for applications where speed is not critical.

Its limitations include:

- Lower data rate compared to synchronous interfaces.
- Reduced tolerance to timing errors, since synchronization depends on software.

The UART (Universal Asynchronous Receiver-Transmitter) protocol is the most representative example of this communication type. It is widely used in RS-232, RS-485, or for communication between a microcontroller and Bluetooth, GPS, or Wi-Fi modules.

VI.2.3 Communication Standards

In embedded electronic systems, communication standards define the rules that allow different components — microcontrollers, sensors, communication modules, and actuators — to exchange information in a reliable, fast, and standardized way. They ensure interoperability between devices, robustness of communication in constrained environments (temperature, vibration, interference), and optimized energy performance.

These standards are divided into:

- Serial communications (SPI, I²C, UART, CAN, USB), and
- Parallel communications (e.g., IEEE 1284).

The choice of standard depends on factors such as required data rate, distance, number of devices, and system constraints in the embedded architecture.

VI.2.4 USB (Universal Serial Bus)

The USB (Universal Serial Bus) is today one of the most widely used communication interfaces in embedded electronic systems. It enables a microcontroller to communicate with a computer, external memory, sensor, or diagnostic module.

This synchronous serial bus is based on a master/slave architecture, where the host (typically a PC or a main microcontroller) initiates and controls all communications with the connected peripherals.

Data transfer occurs through differential signal lines (D+ and D–), which help reduce electromagnetic interference, complemented by a power line (VBUS, typically 5 V) and a ground line (GND).

Recent versions (USB 3.x, USB-C) add extra differential pairs to increase throughput and support new features such as video transmission and fast charging.

Since its introduction in 1996, USB has evolved to meet the growing demands for speed and power in embedded systems. The first generations (USB 1.x) provided speeds up to 12 Mbit/s, sufficient for simple control interfaces such as keyboards or low-data-rate devices.

USB 2.0, still widely used today, raised this rate to 480 Mbit/s, enabling fast data transfers between a microcontroller and a host computer.

Subsequent generations — USB 3.x and USB4 — introduced full-duplex communication and speeds up to 40 Gbit/s, allowing integration into high-performance embedded systems such as industrial cameras, video processing systems, or robotic platforms.

The USB-C connector, now standard in modern architectures, is characterized by its reversibility and versatility: it can simultaneously carry data, power (up to 100 W via Power Delivery), and video signals. Its compact design makes it an ideal choice for portable devices and miniaturized embedded boards.

Version / Type	Maximum Data Rate	Transmission Mode	Available Power	Backward Compatibility	Typical Applications
USB 1.x	1.5–12 Mbit/s	Half duplex	5 V / 100 mA	Yes	Keyboards, simple sensors
USB 2.0	480 Mbit/s	Synchronous	5 V / 500 mA	Yes	Programming, data transfer

Version / Type	Maximum Data Rate	Transmission Mode	Available Power	Backward Compatibility	Typical Applications
USB 3.x	5–20 Gbit/s	Full duplex	5 V / 900 mA	Yes	Industrial cameras, high-speed acquisition
USB4	Up to 40 Gbit/s	Full duplex	5–20 V / 5 A	Yes	High-performance embedded systems
USB-C	Variable depending on standard	Multifunction (data, power, video)	Up to 100 W	Yes	IoT devices, compact embedded boards

VI.2.5 Modern Connectors for Serial Communication (e.g., RJ45)

In embedded electronic systems, communication connectors play a crucial role: they provide the physical and electrical interface between modules while ensuring reliable data transmission, even under mechanical, thermal, or electromagnetic stress. Modern connectors are designed to meet the increasing demands for miniaturization, robustness, and versatility in today's embedded architectures.

Among the most widely used connectors, the RJ45 (Registered Jack 45) holds a key position due to its compatibility with the Ethernet standard (IEEE 802.3). It enables high-speed differential serial communication between embedded electronic devices.

This connector uses twisted-pair cabling, which significantly improves immunity to electromagnetic interference (EMI) and reduces signal loss over long distances. RJ45 connectors support several Ethernet standards, including:

- Fast Ethernet (100 Mbit/s),
- Gigabit Ethernet (1 Gbit/s), and
- 10 Gigabit Ethernet (10 Gbit/s), used in some industrial and automotive embedded systems.

In modern embedded applications, the RJ45 connector is integrated not only for network communication, but also for supervision, diagnostics, and firmware updates. For instance, a microcontroller or System-on-Chip (SoC) can be connected to an industrial server, programmable logic controller (PLC), or central control unit through an embedded Ethernet interface. This approach offers high transfer reliability, global standardization, and interoperability with most existing infrastructures.

In addition to the RJ45, several other modern serial connectors are widely used in embedded electronics:

- **USB-C**: A compact, reversible connector capable of carrying **data, power, and video** signals. It is increasingly adopted in portable devices, medical embedded systems, and development boards.
- Molex / JST / MicroFit: Miniature connectors commonly used in robots, drones, wireless sensors, and compact boards. They provide reliable connections in environments subject to vibration and temperature variation.
- Circular connectors (M12 or LEMO): Used in industrial and automotive applications, they offer excellent sealing and high mechanical strength.
- RJ11 and RJ12: Smaller than the RJ45, these connectors are sometimes used for RS-232/RS-485 serial interfaces or in low-speed embedded systems.

VI.3 Classification of I/O Interfaces in Embedded Electronics

Input/Output (I/O) interfaces ensure communication between the microcontroller and its hardware environment. They enable the exchange of information, the measurement of physical signals, and the control of actuators. Depending on the nature of the exchanged signal and the communication function they perform, I/O interfaces in embedded systems can be classified into four main categories: digital, analog, industrial/network, and specialized interfaces.

VI.3.1 Digital Interfaces

Digital interfaces transmit information in binary form (0 or 1). They are ubiquitous in embedded systems, particularly for communication between the microcontroller and logical peripherals. These interfaces are divided into three main subcategories:

- Simple interfaces: General Purpose Input/Output (GPIO) pins are individually configurable as inputs or outputs and are used to read or drive simple logic signals. PWM (Pulse Width Modulation) is a digital output with modulated pulse width, commonly used to control motor speed or light intensity.

- Serial communication interfaces: These transmit data bit by bit, reducing the number of required lines. The most common ones include SPI, I²C, UART, CAN, and 1-Wire.
- Parallel interfaces: Used to transfer multiple bits simultaneously, they offer very high data throughput at the cost of more complex wiring. Two types can be distinguished: Internal buses (e.g., AHB, AXI, APB) for communication between the processor, memory, and peripherals. External buses such as IEEE 1284, used for high-speed short-distance transfers.

VI.3.2 Analog Interfaces

Analog interfaces enable the embedded system to interact with the real world, where physical quantities such as temperature, pressure, voltage, and current vary continuously. They perform conversion between analog and digital signals using dedicated circuits:

- ADC (Analog-to-Digital Converter): Converts an analog signal into a digital value, allowing the microcontroller to perform numerical processing.
- DAC (Digital-to-Analog Converter): Generates an analog signal from a digital value, typically used to drive proportional actuators or generate waveforms.

These interfaces are essential in measurement, control, and analog regulation systems, such as temperature sensors, microphones, pressure sensors, and amplifiers.

VI.3.3 Industrial Communication Interfaces and Networks

These interfaces handle communication between multiple embedded systems or between an embedded system and industrial machinery. They ensure reliable transmission, often over long distances, thanks to standardized protocols. The main types include:

- RS-232 / RS-485: Asynchronous or differential serial interfaces, well suited to robust industrial links.
- Modbus / Profibus: Standardized industrial automation and supervision protocols.
- Ethernet / RJ45: High-speed network communication for connected systems (e.g., IoT, SCADA).
- USB (Universal Serial Bus): A universal interface for communication, diagnostics, and peripheral power supply.

These interfaces are commonly used in programmable logic controller (PLC) networks, inter-board communication, or supervisory and maintenance systems.

VI.3.4 Specialized and Debugging Interfaces

These interfaces are designed for specific purposes, particularly for programming, testing, and diagnostics of embedded systems. They provide direct access to the microcontroller core for configuration or firmware updating. The most common include:

- JTAG (Joint Test Action Group): A standardized interface for testing, programming, and debugging integrated circuits.
- SWD (Serial Wire Debug): A simplified variant of JTAG, widely used in ARM microcontrollers.
- ISP (In-System Programming): Allows reprogramming of the microcontroller without removing it from the circuit board.

These interfaces are indispensable during development, maintenance, and production phases, ensuring precise diagnostics and easy system updates.

Liste des abréviations

CAN	Convertisseur Analogique Numérique
DC	Direct Current
E/S	entre / Sortie
EEPROM	Electrically Erasable Programmable Read-Only Memory
FTDI	Future Technology Devices International
Gnd	ground
I/O	Input/Output
I2C	Inter-Integrated Circuit
IDE	Integrated Development Environment
IoT	Internet of Things
LCD	Liquid Crystal Display
LED	Light-Emitting Diode
MISO	Master In Slave Out
MOSFET	Metal Oxide Semiconductor Field Effect Transistor
MOSI	Master Out Slave In
PWM	Pulse Width Modulation
RF	radio frequency
RFID	Radio-Frequency Identification
RISC	Reduced Instruction Set Computer
SCK	Serial Clock
SCL	Serial Clock
SDA	Serial Data
SPI	Serial Peripheral Interface
SRAM	Static Random Access Memory
SS	Slave Select
USART	Universal Synchronous & Asynchronous Receiver Transmitter
USB	Universal Serial Bus
Wi-Fi	Wireless Fidelity

Références

MEGNAFI, Hicham et MERAD, Lotfi. Mesure et instrumentation-Capteurs. 2024.

MEGNAFI, Hicham. Système à micro contrôleur–EasyPICV7. 2024.

MEGNAFI, Hicham. Électronique Embarquée–Arduino. 2024.

BRAHAMI, Mustapha Anwar et MEGNAFI, Hicham. Polycopie TP Réseaux et Protocoles. 2022.

ABDELLAOUI, Ghouti et MEGNAFI, Hicham. Systèmes embarqés et temps réel (RaspberryPi). 2019.

MEGNAFI, Hicham, ABDELLAOUI, Ghouti, et MR BRAHAMI, Mustapha Anwar. Systèmes à Microcontrôleur. 2019.

John Boxall ,Arduino Workshop: A Hands-On Introduction with 65 Projects.2013.

Mark Geddes, Arduino Project Handbook: 25 Practical Projects to Get You Started.2016.

Pierre-Yves Rochat, EPFL.introduction au microcontrôleur. Ecole Polytechnique fédérale

de Lausanne. 2016

Massimo Banzi ,Getting Started with Arduino2015.

H. Megnafi, A. Ayad, W. Tabib, A. A Mouaziz, R. Ould Babaali, I. Medjhoud, Improvedprinting time by changing the mechanical part of the 3D printer, embedded systemapplication, NewMat'21–1st International Conférence: New Trends on InnovativeConstruction Materials-ESSA-Tlemcen (Algeria)–22, 23 March, 2022.

M.A. Brahami, H. Megnafi, H. Boukeffous, O. Benlaldj, Design and implementation of anembedded system for effective attendance management, NewMat'21–1st InternationalConférence: New Trends on Innovative Construction Materials-ESSA-Tlemcen(Algeria)–22, 23 March, 2022.

MEGNAFI, Hicham, KARAOUZENE, Zoheir, MERAD, Lotfi, et al. Internet of Things technology for efficient fire hydrant management. In : 2023 IEEE International Workshop on Mechatronic Systems Supervision (IW_MSS). IEEE, 2023. p. 1-6.