الجمهورية الجزائرية الديمقراطية الشعبية

Democratic and Popular Republic of Algeria

وزارة التعليم العالي و البحث العلمي

Ministry of Higher Education and Scientific Research

جامعة عمار ثليجي - الأغواط

Faculty: Science

Department: Computer Science

Field : Informatics

Speciality : Computer Science Engineering

Handout of (Lecture, Tutorials, Labs)
Intended for students of: 1st Year Common Core Level Engineer
in Computer Science,

Computer Architectures

Authored by : Dr. Leila BENAROUS

MCA, University of Laghouat

Email : l.benarous@lagh-univ.dz

University year : 2025/2026

Table of Contents

List of Figures

Lists of Tables

General Introduction

This textbook is designed to provide first-year Computer Science and Engineering students with a comprehensive understanding of computer architectures. It serves as a foundation for studying the fundamental principles that manage the design and operation of modern computing systems.

In this book, we explore the architecture of computers by exploring the internal organization of their central processing units (CPU), memory systems, and I/O components. The subject of computer architecture is essential in the field of computer science and engineering, as it directly influences the performance, efficiency, and capability of computing systems.

This textbook is structured to provide a gradual and coherent progression through the subject matter. Each chapter is organized to build upon previous topics, ensuring a solid foundation for more advanced concepts. Starting with the basic organization of a computer's central unit, we proceed to explore the internal architecture of processors, including their components, design, and instruction execution models. The book also introduces historical developments in processor architectures, tracing the evolution from early systems to modern 64-bit and multicore processors.

In subsequent chapters, we study specific examples, such as the Intel 8086 processor, and provide detailed discussions of key architectural concepts, including instruction sets, addressing modes, pipelining, and memory models. We also address the latest advancements in computer architecture, such as the impact of multicore processors, cache memory systems, and virtual memory, providing students with a broad view on current and future trends in the field.

By the end of this textbook, students will be able to:

- Understand the fundamental components of a computer system, including the CPU, memory hierarchy, and I/O systems.
- Grasp the concepts behind processor design, including instruction set architecture, addressing modes, and the various execution steps within a processor.
- Distinguish between different processor architectures, including RISC and CISC, and understand their implications for performance and efficiency.
- Analyze the role of cache memory, memory hierarchies, and virtual memory in optimizing system performance.
- Learn about modern advancements in computer architecture, including multicore systems, AI accelerators, and quantum computing.

CHAPTER 1: General Organization of a Computer's Central Unit

I.1. History of Computers

First Generation (1940-1956): Used vacuum tubes (see figure 1.1 for illustration) for circuitry and magnetic drums for memory (see figure 1.2 for illustration). Example: ENIAC (refer to figure 1.3 for illustration), UNIVAC (see figure 1.4 for illustration).

Figure 1. 1 : Vacuum Tubes [1]

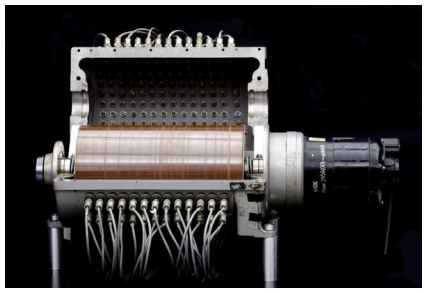

Figure 1. 2 : Magnetic Drum memory Electronic[2]

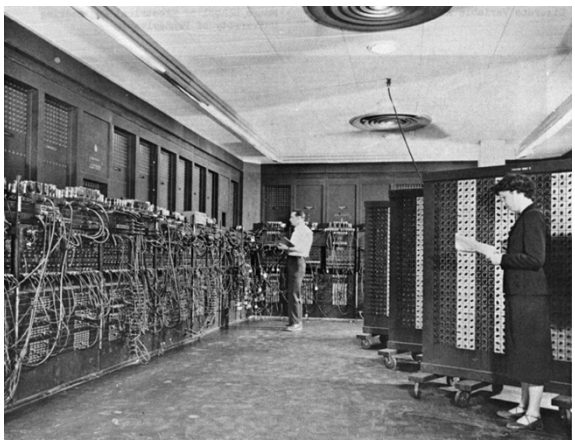

Figure 1. 3 : Numerical Integrator and Computer (ENIAC) [3]

Figure 1. 4 : UNIVersal Automatic Computer (UNIVAC)[4]

Second Generation (1956-1963): Used transistors (Figure 1.5), which were smaller, faster, and more reliable than vacuum tubes. Refer to Figure 1.6 exhibiting the first experimental transistor-based computer.

[1] https://images.app.goo.gl/iUKcThRYpbiM7dYi9

[2] https://images.app.goo.gl/Jk7gx2bUX2M8e9wa8

[3] https://images.app.goo.gl/KDrB35ZKuQVmaNPB9

[4] https://images.app.goo.gl/c6WtJ5iumHpxUknV8

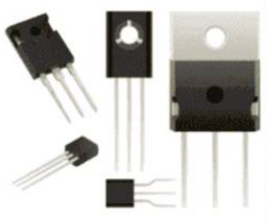

Figure 1. 5 : Transistors

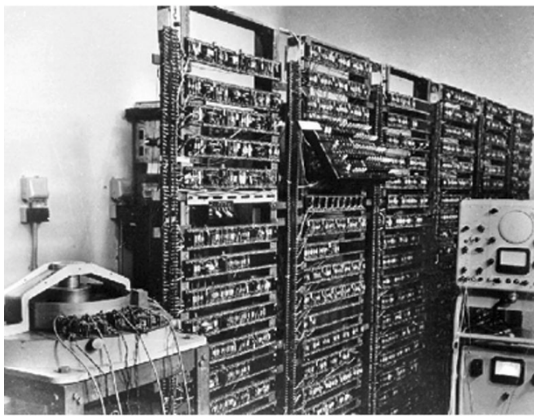

Figure 1. 6 : Manchester's Experimental Transistor Computer[5]

Third Generation (1964-1971): Introduced integrated circuits (Figure 1.7), further reducing size and increasing speed.

Refer to Figure 1.8 exhibiting an integrated circuit-based computer.

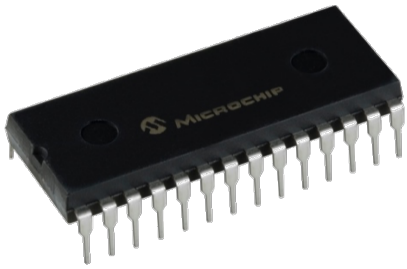

Figure 1. 7 : Integrated circuits [6]

Figure 1. 8 : integrated circuits-based Computer[7]

Fourth Generation (1971-Present): Features microprocessors (Figure 1.9), allowing entire CPU functions to fit on a single chip.

[5] https://images.app.goo.gl/XdQTQ4jwzKDpRcSFA

[6] https://images.app.goo.gl/nRXEB3rcNQA1pnxz5

[7] https://images.app.goo.gl/6vi8ZqcKGWDy4Pon8

Figure 1. 9 : Microprocessors

Fifth Generation (Present and Beyond): Focuses on artificial intelligence (Figure 1.10), quantum computing (Figure 1.11), and advanced parallel processing (Figure 1.12).

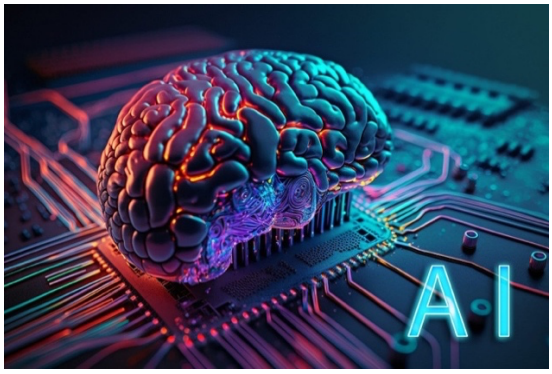

Figure 1. 10 : AI-based computer[8]

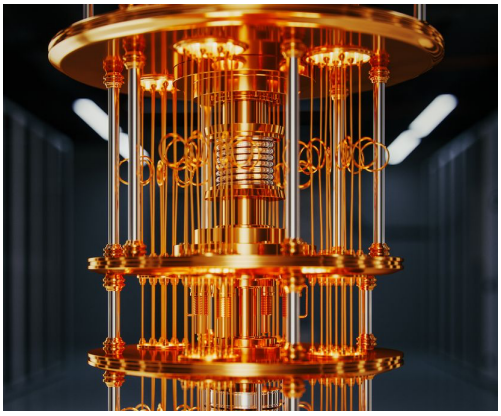

Figure 1. 11 : quantum computer [9]

Figure 1. 12 : super calculator[10]

[8] https://images.app.goo.gl/n1HUJ1xhL2Vrqkqz8

[9] https://images.app.goo.gl/ZmQyvmjgCJPoh9bu7

[10] https://images.app.goo.gl/if3BhW69HLCLuuyh6

I.2. Fundamental Concepts of a Computer

A computer is an electronic device that processes data based on predefined instructions. Key functions include:

- **Input**: Receiving data through input devices (keyboard, mouse, scanner).
- **Processing**: Executing instructions using the processor.
- **Storage**: Saving data temporarily or permanently in memory or storage devices.
- **Output**: Displaying processed results through monitors, printers, or speakers.

I.3. Essential Components of a Computer

The computer is composed of multiple electronic components that handle each a specific task. Among these components are the processor, the co-processors, the different types of memory (cache, main and secondary), in addition to the input/output peripherals. Figure 1.14 illustrates the evolution of processor from the first one "Intel 4004" used in calculator to the recent releases with billions of integrated transistors.

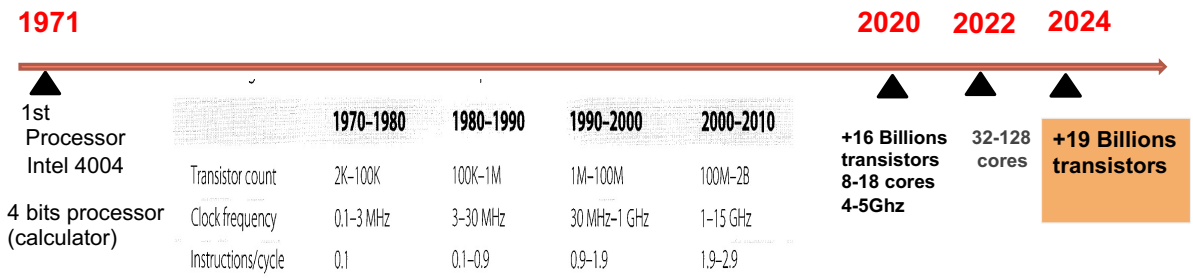

Figure 1. 13 : Processor's evolution

- **Processor (CPU):** The brain of the computer, responsible for executing instructions.
- **Co-Processors** are specialized processing units that work alongside the main CPU to handle specific tasks more efficiently. Co-processors offload complex calculations from the CPU, improving performance in specialized areas. Examples of co-processors are Floating-Point Unit (FPU), Graphics Processing Unit (GPU), Cryptographic Co-Processor, and Neural Processing Unit (NPU).
- **Memory:** Includes **RAM** (temporary storage), **ROM** (permanent storage for firmware), Cache and Virtual. Figure 1.15 depicts the hierarchy of memory starting by the fastest type and smallest in size which are registers to the slowest type of memory and with the largest storage capacity such as the tapes. In the figure, as we go from top to down, the memory size increases and the speed decreases.

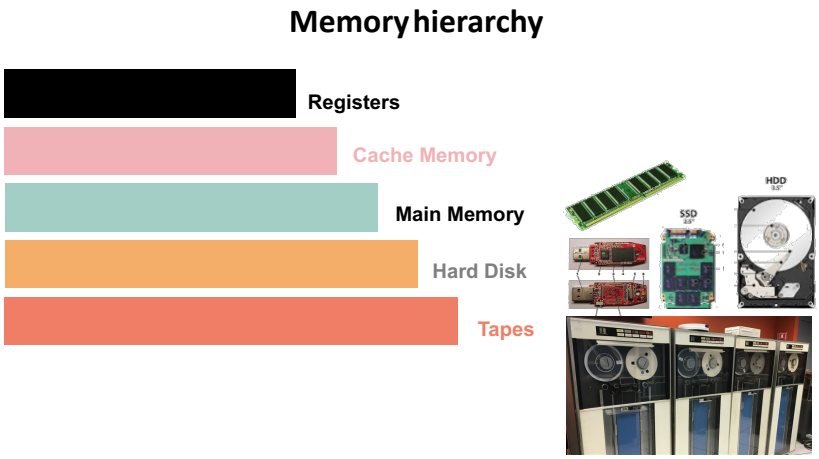

Figure 1. 14 : Memory Hierarchy

- **Input Devices:** Devices used to provide data (keyboard, mouse, microphone). Refer to Figure 1.16 for illustration of common input peripherals.

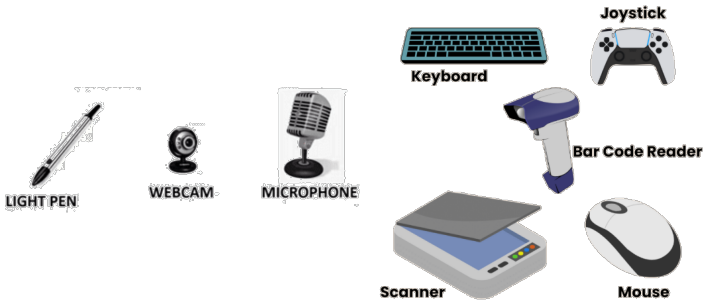

Figure 1. 15 : input devices [11]

- **Output Devices:** Devices that present results (monitor, printer, speakers). Refer to Figure 1.17 for illustration of common output peripherals.

Figure 1.18 depicts the peripherals that can be input and output devices at the same time in addition to the peripherals that can be used either as input or output.

Figure 1. 16 : Output devices [12]

[11] https://images.app.goo.gl/8vzED7qumpL3UehXA

[12] https://images.app.goo.gl/pvKcvVSdFCZcZkqd8

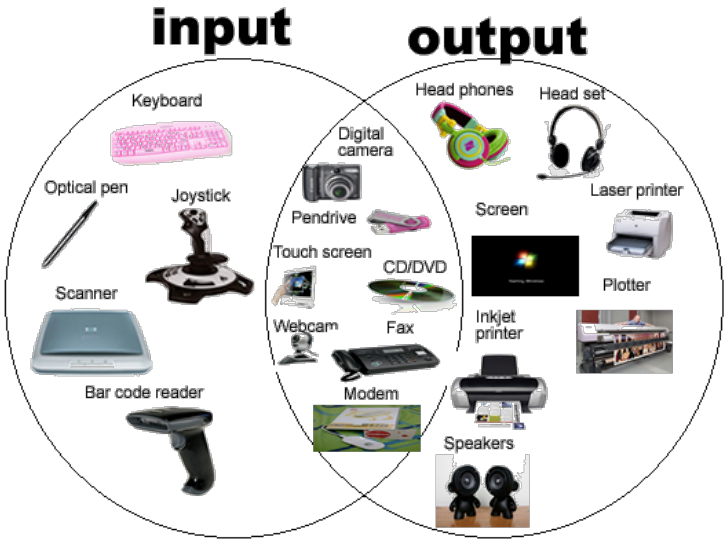

Figure 1. 17 : input/output devices [13]

I.4. Basic Computer Architectures

For the design of computers, two architectures have been in use, the first is the Von Neumann Architecture and the second is the Harvard architecture. Although the second architecture is older, the first one is the most commonly used in general purpose computers mainly due to production costs.

Von Neumann Architecture is a digital computer architecture whose design is based on the concept of stored program computers where program data and instruction are stored in the same memory (see figure 1.19). This architecture was designed by the famous mathematician and physicist John Von Neumann in 1945.

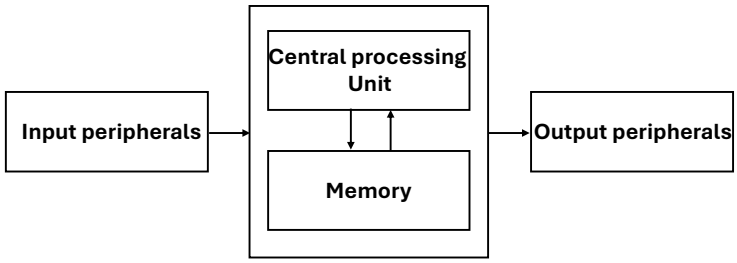

Figure 1. 18 : Von Neumann Architecture

Harvard Architecture is the digital computer architecture whose design is based on the concept where there are separate storage and separate buses (signal path) for instruction and data. Thus, it overcomes by design the bottleneck issue of Von Neumann Architecture. (see figure 1.20 for illustration)

[13] https://images.app.goo.gl/dRZXbh7LvjmboSco6

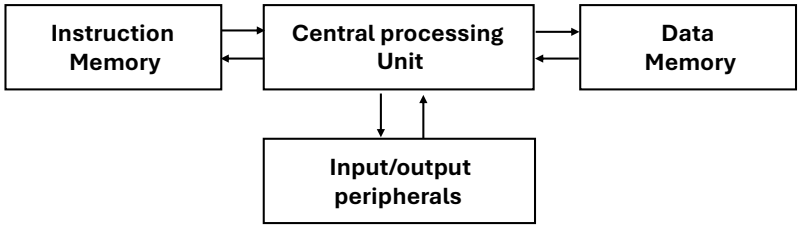

Figure 1. 19 : Harvard Architecture

Table 1. 1: Difference between Von Neumann and Harvard Architecture

Von Neumann Architecture	**Harvard Architecture**
- Same physical memory address is used for instructions and data. - There is common bus for data and instruction transfer. - Two clock cycles are required to execute single instruction. - It is cheaper in cost. - CPU cannot access instructions and read/write at the same time. - It is used in personal computers and small computers.	- Separate physical memory address is used for instructions and data. - Separate buses are used for transferring data and instruction. - An instruction is executed in a single cycle. - It is costly than VN Architecture. - CPU can access instructions and read/write at the same time. - It is used in micro controllers and signal processing.

I.5. Conclusion

Modern computers primarily follow the **Von Neumann architecture**, but with certain elements inspired by the **Harvard architecture**.

They use:

- **Separate** instruction and data **caches** (like **Harvard**)
- **Unified main memory** (like **Von Neumann**) and to solve the bottleneck issue: Modern memory architectures use **multi-channel memory** to increase bandwidth and allow more data to flow between memory and the CPU in parallel.

I.6. QCM

1. Which of the following is considered the first mechanical computer?

A) ENIAC

B) UNIVAC

C) Difference Engine

D) IBM 701

2. Who is known as the "Father of the Computer"?

A) Bill Gates

B) Charles Babbage

C) Alan Turing

D) John von Neumann

3. What does CPU stand for?

A) Central Processing Unit

B) Control Program Unit

C) Computing Peripheral Unit

D) Central Programming Utility

4. Which generation of computers used vacuum tubes?

A) First generation

B) Second generation

C) Third generation

D) Fourth generation

5. The basic operations performed by a computer include:

A) Input, Processing, Output, Storage

B) Input, Calculation, Printing, Networking

C) Data Entry, Internet Access, Graphics, Sound

D) None of the above

6. Which component fetches instructions from memory and decodes them?

A) Memory

B) Arithmetic Logic Unit (ALU)

C) Control Unit

D) Hard Drive

7. What is the main function of the ALU?

A) To control input/output devices

B) To store data permanently

C) To perform arithmetic and logical operations

D) To manage memory allocation

8. Which of the following is NOT an essential component of a computer?

A) CPU

B) RAM

C) Monitor

D) Printer

9. In which year was the first electronic general-purpose computer (ENIAC) completed?

A) 1943

B) 1946

C) 1950

D) 1952

10. What is the primary purpose of cache memory?

A) To store permanent data

B) To speed up access to frequently used data

C) To back up the hard drive

D) To replace RAM when it fails

11. Which type of architecture is most commonly used in modern computers?

A) Harvard Architecture

B) Von Neumann Architecture

C) RISC Architecture

D) CISC Architecture

12. In the Von Neumann model, what is stored in the same memory?

A) Programs and data

B) Hardware drivers and applications

C) Operating system and user files

D) None of the above

13. Which of the following is a secondary storage device?

A) RAM

B) ROM

C) Hard Disk Drive

D) Cache

14. What is the full form of RAM?

A) Random Access Memory

B) Read Access Module

C) Real Available Memory

D) Rapid Application Memory

15. Which part of the CPU performs multiplication and division?

A) Control Unit

B) Memory Management Unit

C) Arithmetic Logic Unit

D) Instruction Decoder

16. Which of the following is a pointing device?

A) Keyboard

B) Mouse

C) Printer

D) Scanner

17. Which computer architecture uses separate memories for instructions and data?

A) Von Neumann

B) Harvard

C) RISC

D) CISC

18. Which of the following is NOT a characteristic of primary memory?

A) Fast access

B) Volatile

C) Permanent storage

D) Expensive compared to secondary storage

19. Who proposed the stored-program concept?

A) Charles Babbage

B) Blaise Pascal

C) John von Neumann

D) Ada Lovelace

20. Which of the following best describes a bus in computer architecture?

A) A type of memory

B) A software program

C) A set of physical connections for data transfer

D) An output device

I.7. QCM Solutions

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
C	B	A	A	A	C	C	D	B	B	B	A	C	A	C	B	B	C	C	C

Chapter 2: Internal Architecture of Processors

II.1. Introduction about Processor design

The Central Processing Unit (CPU), or processor, is the primary component of a computer system responsible for executing instructions and performing computations. As illustrated in the previous chapter, the processor is made of integrated circuits mainly made of transistors. These transistors increased throughout the years, in an aim to enhance the performance of computers. According to Gordon Moore, one of the founders of Intel, **"Number of integrated devices in a single piece of Silicon will double roughly every 18-24 months".** This prediction proofed to be right ever since it was announced, and it is often referred to as the Gordon Moore Law. Figures 2.1 and 2.2 illustrate the intel 4004 circuit scheme and architecture respectively. Figure 2.3 depicts the intel 4004 processor.

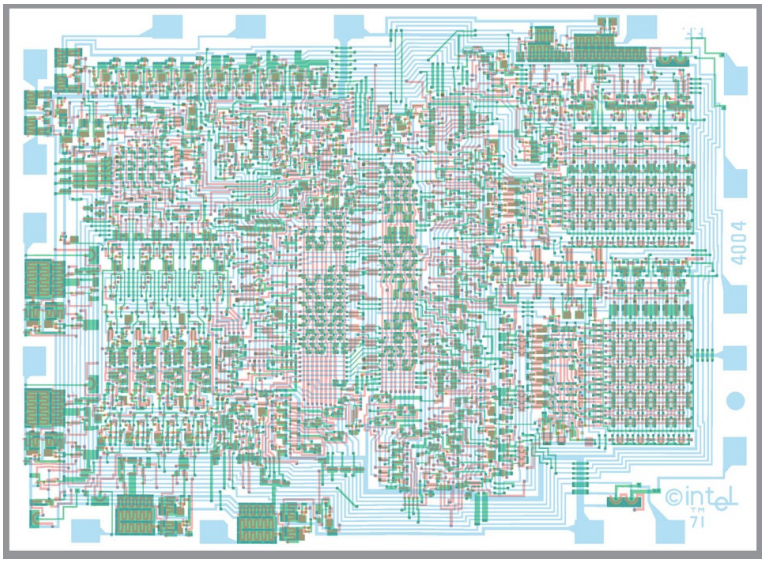

Figure 2. 1: Intel 4004 Circuit scheme [14]

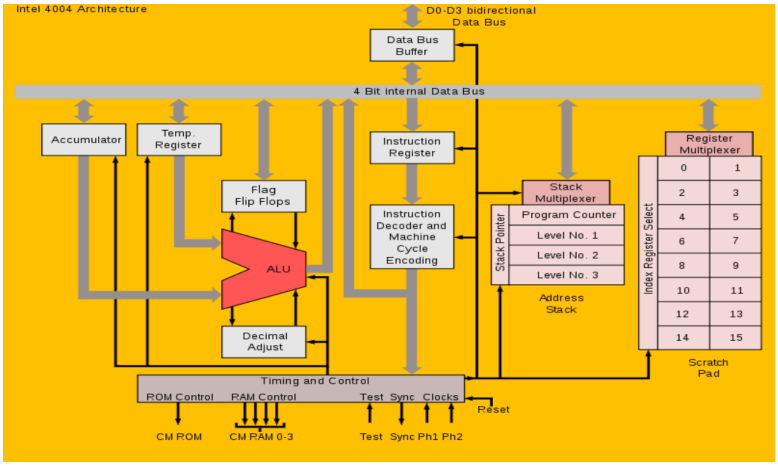

Figure 2. 2: Intel 4004 Architecture [15]

[14] https://images.app.goo.gl/Yrx7VmVDp4qWBx6d7

[15] https://images.app.goo.gl/oCJznRuDyZMF1KbM7

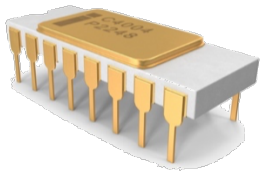

Figure 2. 3: Intel 4004 microprocessor[16]

The design process of a processor is cyclic, and it starts by setting out the specifications, which are the set of instructions and commands that the processor being designed is supposed to do when produced. The next step in the process is to attempt to find the appropriate implementation for the made design, this phase is denoted as synthesis phase. this may include searching for appropriate materials to use, selecting the engraving techniques, deciding the components and their locations, etc. all this while considering the technological limits and attempting to overcome them such as the impact of heating, the fine engraving and others. The cost is another decisive criterion when working on this phase. The subsequent phase is the implementation which results in producing the processors hardware. The last phase is to analyze the produced processor to ensure that it function accurately and effectively as it is expected from it. Each processor is made to function in a certain condition such as within an interval of threshold lowest and highest temperatures. The analysis is done through testing benchmarks to verify that the computations are correct and executed with the expected speed, it also includes the inspection of the processor's hardware to guarantee that all of its components are functioning without failure. The process is cyclic because the processors makers are continuously working of producing improved versions to match the evolution of applications. Figure 2.4 illustrates the process.

[16] https://images.app.goo.gl/dHzqmtwsbAkCAkEu9

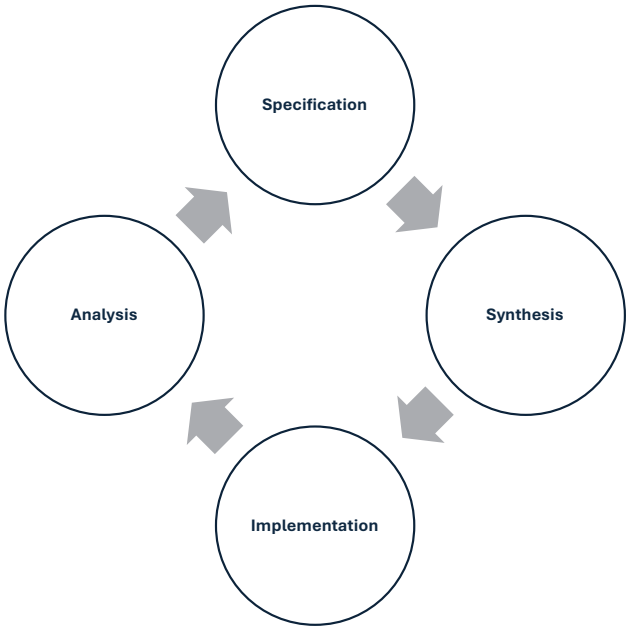

Figure 2. 4: Microprocessors Design Process

II.2. Microprocessor Components

The legacy processor was composed of key components as depicted in Figure 2.5. The modern processors include these components besides others and have more complex design that allows it to do advanced types of computations. For the simplified design of microprocessors, it includes arithmetic logic unit (ALU), registers, control unit (CU) besides instruction, data and address buses.

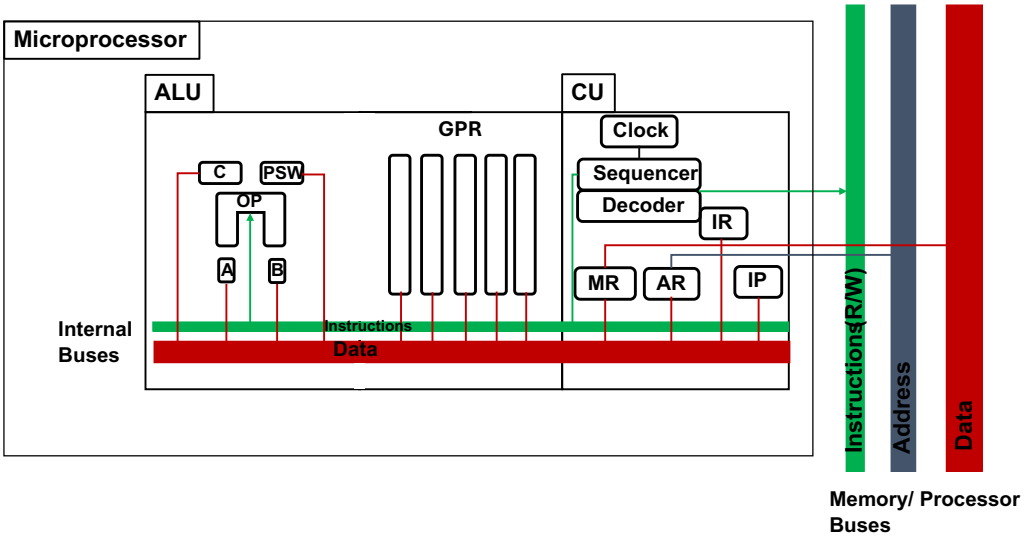

Figure 2. 5: Microprocessors key components (simplified)

A,B and C: Operators (registers); **OP**: Operation;

II.2.1. Calculation unit

The calculation unit is responsible for executing the different types of the arithmetic, logic and floating points computations on various types of scalar and vectorial data. It includes the arithmetic logic unit, the floating-point unit and the multimedia unit.

- **Arithmetic and Logic Unit(s):** The ALU is the computational core of the processor, responsible for performing arithmetic and logical operations. It executes basic arithmetic operations (addition, subtraction, multiplication, division). Performs logical operations (AND, OR, NOT, XOR). Conducts comparison operations (e.g., checking if two values are equal).
- **Floating Point Unit (s)**
- **Multimedia Unit (s) (vectorial calculations)**

II.2.2. Control Unit (CU)

It coordinates the Processor's components to execute a set of instructions or a program. It is composed of a decoder, a sequencer, a clock and two registers. The registers are the instruction register (IR) and the instruction pointer - aka counter (IP). The CU uses two other registers which are Address Register (AR) and Memory Register (MR). The AR stores the address of a memory word, and the MR stores the content of a memory word.

The **clock** defines the basic cycle used to synchronize each step of the research and the execution cycles.

Noting that the **CPU Cycle** is the minimal time used to **execute an instruction** including both of its phases (search and execution).

The **sequencer** is responsible for controlling the order of micro-instructions execution, it can be both **wired** (hardware) or **firmware** (micro-programmed). The **wired** sequencer is faster but harder to make. The **firmware** sequencer is easier to make and more flexible.

The CU ensures that instructions are executed in the correct sequence. It handles branching and jumping by updating the Program Counter (PC). The CU is essential for implementing pipelining and parallelism in modern processors.

The CU is responsible for the following tasks**:**

- **Instruction Fetch**: Retrieves instructions from memory using the Program Counter (PC).
- **Instruction Decode**: Interprets the fetched instruction and determines the required operations.

- **Execution Control**: the CU Generates control signals to coordinate the ALU, registers, and memory. Also, it manages data flow between components.

II.2.3. Buses

Data, addresses, and instructions (control signals) travel between the processor's components through internal buses. To prevent buses from becoming a bottleneck and slowing down the system, buses are designed to be wide (bus size ≥ register size) and often include multiple lines to increase bandwidth.

II.2.4. Registers

A register is a **word memory** within the **processor.** The processor includes multiple types of registers mainly grouped into functioning registers which are used by the processor and cannot be controlled by the programmer and general-purpose registers which are used by the processor and can be managed by the programmer as well.

Functioning Registers also known as **special-purpose** registers englobe:

- **Instruction Pointer** (or Program Counter): It contains the address of the next instruction to fetch from memory after successfully finishing the execution of the current one.
- **Instruction Register**: It contains the address of instruction that is just about to be executed.
- **Accumulator**: It is found in different numbers within the ALU of the processor. It stores Operand/result before it's saved in memory.
- **Memory Data Register (MDR or MR):** contains the data to be read from or written in the address location.
- **Memory Address Register**: contains the address of the location to be accessed
- **Index Registers (XR):** used to parse the tables easily. They can be incremented or decremented.
- **Stack Pointer**: It simulates the stack in the memory. It stores the address of the last item put into the stack (Last In First Out).
- **Program Status Word:** a set of flags (bits) representing a particular status. Such as: **C: c**arry flag**; Z: z**ero flag**; O: o**verflow flag. They are used to test results in branches (**Jump**)

General Purposes Registers: They are accessed by programmers of assembly language. They either save temporal results or frequently used ones without having to store and fetch them back and forth every time. Therefore, the GPR usage reduces the execution time making the processor performance faster and more efficient.

II.3. Instruction Set Architecture

Microprocessors are instruction set processor (ISP) that executes instructions from predefined instructions set. All programs that run on a microprocessor are encoded in that instruction set which is known as the instruction set architecture (ISA). In the design methodology, an ISA is the specification of a design (behavioral description). While the ISP is the implementation of this design (structural description). ISA defines a set of instructions called Assembly language. ISA have been differentiated according to the number of operands that can be explicitly specified in each instruction. Examples:

- 2 addresses architecture
- 3 addresses architecture
- Accumulator-based architecture
- Stack-based architecture
- Modern ISA assumes that operands are stored **in multi-entry register file**.

ISA plays crucial roles. First, it represents contract between a software and a hardware (see Figure 2.6). Moreover, it serves as the specification of a microprocessor design. Also, it defines an interface that separates what is done **statically** at the **compile** time **VS** what is done **dynamically** at a **run** time.

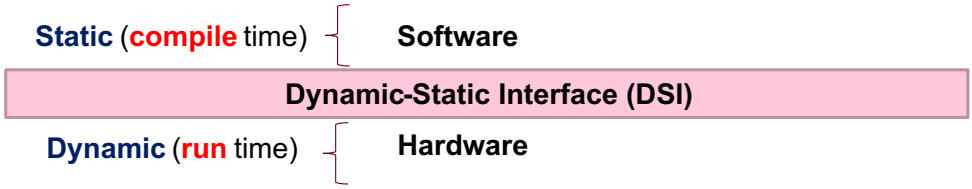

Figure 2. 6: ISA-DSI

Categories of Instructions defined in ISA:

- **Data Transfer Instructions:** Move data between memory and registers. Examples: LOAD, STORE.
- **Arithmetic Instructions:** Perform mathematical operations. Examples: ADD, SUBTRACT, MULTIPLY.
- **Logical Instructions:** Perform logical operations. Examples: AND, OR, NOT.
- **Control Flow Instructions:** Alter the sequence of instruction execution. Examples: JUMP, BRANCH, CALL.
- **Input/Output Instructions:** Facilitate communication with external devices. Examples: IN, OUT.

II.4. Addressing Modes

Addressing modes define how the processor accesses operands for instructions. They play a crucial role in determining the flexibility and efficiency of instruction execution.

- **Immediate Addressing:** The operand is specified directly in the instruction. Example: ADD R1, #5 (Add 5 to register R1).
- **Direct Addressing:** The address of the operand is specified in the instruction. Example: LOAD R1, 1000 (Load data from memory address 1000 into R1).
- **Indirect Addressing:** The address of the operand is stored in a register or memory location. Example: LOAD R1, (R2) (Load data from the address stored in R2).
- **Indexed Addressing:** The operand address is calculated by adding an index value to a base address. Example: LOAD R1, 1000(R2) (Load data from address 1000 + value in R2).
- **Relative Addressing:** The operand address is relative to the current instruction pointer. Example: JUMP +10 (Jump 10 instructions ahead).

II.5. Instruction Execution Steps

The execution of instructions in a processor follows a well-defined cycle, known as the Instruction Cycle. This cycle consists of five main phases:

- **Fetch:** The Control Unit retrieves the instruction from memory using the Program Counter (PC). The PC is incremented to point to the next instruction.
- **Decode:** The Control Unit interprets the fetched instruction and determines the required operations.
- **Load operand:** loads operand from memory
- **Execute:** The ALU performs the necessary calculations or operations. Data is moved between registers and memory as needed. Noting that the execution cycle differs if the instruction to execute is **"Jump"**, no calculation done, the content of MR will be placed in IP. **Similarly,** if the instruction is "**write**", the content of MR will be placed in memory, no calculation done.
- **Write (Store):** The result of the operation is stored in a register or memory location.

II.6. RISC and CISC architectures

Modern processor designs are typically categorized under two major architectural paradigms: RISC (Reduced Instruction Set Computer) and CISC (Complex Instruction Set Computer). These two approaches represent fundamentally different philosophies in instruction set design, execution efficiency, and hardware complexity. RISC emphasizes simplicity and speed, using a small, highly optimized set of instructions. CISC emphasizes functionality and compactness, providing more complex instructions that can perform multi-step operations. Both architectures have evolved over time, and many modern CPUs incorporate elements of both, but understanding their original design goals remains essential to grasping processor behavior.

II.6.1. RISC: Reduced Instruction Set Computer

The RISC architecture is characterized by its use of registers through its (3,0) load and store memory access model. As such, its instructions have **fixed size** and **format**. RISC has **a smaller** number of instructions, the reduced number is not about the type of instructions being less, it is due to their representation. Early CISC processors typically had fewer general-purpose registers (e.g., 8), whereas RISC processors commonly provide around 32 GPRs. This larger register set is necessary because, in RISC architectures, all arithmetic and logic operations are performed using registers, with memory accessed only through load and store instructions. This makes the pipelining (data parallelism) possible in RISC architecture. Therefore, one program in RISC may need **more instructions** to execute (than in **CISC**). However, these instructions are **simpler**, **easier** and **faster** to execute than those of **CISC**. An example of RISC based processors is **IBM Power PC processor**

II.6.2. CISC: Complex Instruction Set Computer

In the CISC architecture, instructions have a **non-fixed size**. It is complex because the number of instruction is large, not in their type but in their representation, for example an addition instruction can be represented via various circuits to execute it, the one that use registers only such as: **Add** R, R, R, the one that uses memory access only like **Add** A, A, A; or, the one that mixes the use of registers and memory access as **Add** R, A, A. Noting that **R** refers to register **and A** is for memory address. Since this architecture allows multiple representations, the execution time of a calculation instruction is not fixed as it depends on whether it is having a direct memory access of using registers. The variable instruction execution time makes the pipeline hard to achieve is CISC. From the programmers' point of view, the CISC architecture offer simpler more flexible instructions. An example of CISC processors are **Intel** and **AMD**

II.7. Memory Models

The processor executes instructions on operands. A Simple Instruction example: A=B+C can be executed differently based on the way operands are accessed and fetched from memory, directly, by use GPRs, through the use of ALU accumulators, etc. we call these, memory access models, to explain them, we use the previous example of addition. For the following, we use this notation (m, n), where m is the maximum number of operands per instruction, and n is the maximum number of operands per instruction that are directly accessed from memory.

Back to our previous example: A=B+C

- In the **(3,3) model** aka Memory-Memory Model (refer to Figure 2.7.A)

  Add @A, @B, @C

- In the **(1,1) model** aka Memory-Accumulator Model (refer to Figure 2.7.B)

  Load @B —> Acc:= B

  Add @C -> Acc:=Acc + C

  Store @A -> A:=acc

- In the **(2,1) model** aka Memory-Register Model (refer to Figure 2.7.C)

  Load R1, @B —> R1:= B

  Add R1, @C -> R1:=R1+C

  Store R1, @A -> A:=R1

- In the **(2,0)** model direct calculation using registers (special case of (2,1) model)

  Add R1, R2

- In the **(3,0) model** aka Register-Register Model or Load/Store Model (used in RISC Processors) (refer to Figure 2.7.D)

  Load R1, @B

  Load R2, @C

  Add R3, R1, R2

  Store @A, R3

- In the **(0,0) model** aka Stack Model (refer to Figure 2.7.E)

  Push @B

  Push @C

Add

Pop @A

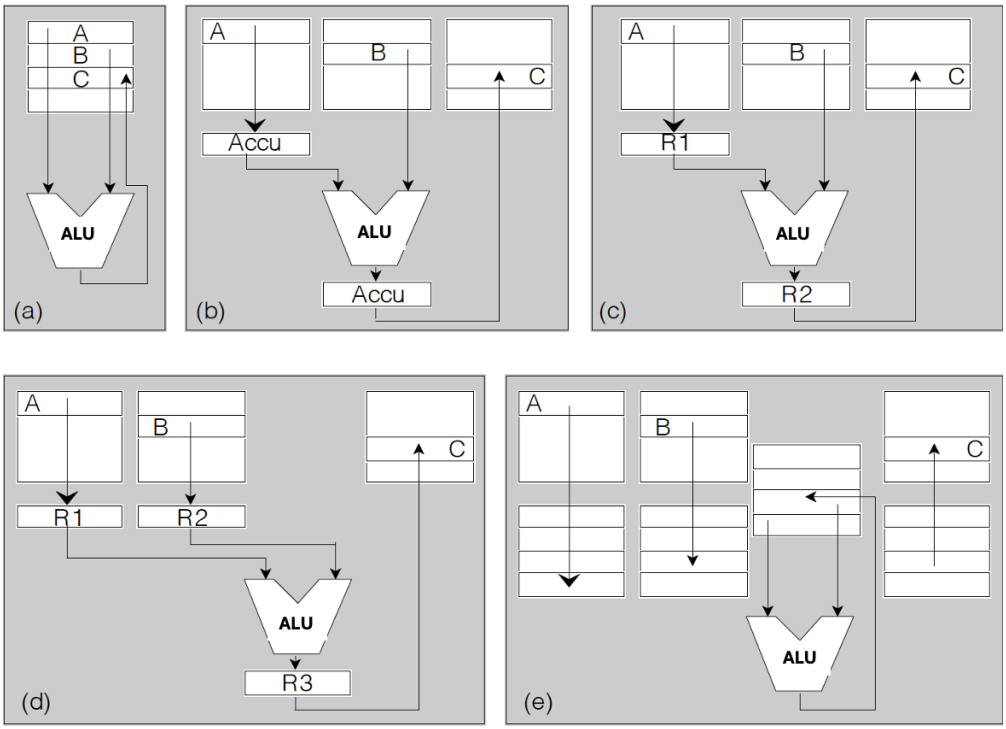

Figure 2. 7: Memory access models

II.8. Flynn's classification for computer architectures

- **SISD (Single Instruction Single Data) architectures** are the simplest and most common type of architecture. They are designed to execute one instruction on one data element at a time. This makes them well-suited for sequential applications.
- **SIMD (Single Instruction Multiple Data) architectures** are more complex than SISD architectures and can execute one instruction on multiple data elements simultaneously. This makes them much more efficient at handling parallel tasks such as video processing or image rendering.
- **MISD (Multiple Instruction Single Data) architectures** are even more complex than SIMD architectures and can execute multiple instructions on one data element simultaneously. This makes them very well suited for highly parallel applications such as scientific computing or database management.
- **MIMD stands for Multiple Instruction, Multiple Data**. It is a type of computer architecture where multiple processors are used to execute multiple instructions on multiple pieces of data. Refer to Figure 2.8 for an illustration.

SISD **(Single Instruction, Single Data)** Traditional Von Neumann single CPU computer (sequential execution)	**MISD** **(Multiple Instruction, Single Data)** Pipelined Computer
SIMD **(Single Instruction, Multiple Data)** Vector Processor	**MIMD** **(Multiple Instruction, Multiple Data)** Multi-cores Multi-processors Multi-computers

Figure 2. 8: Flynn's computer architecture classification

II.9. Conclusion

In this chapter, we viewed the computer architecture with its different types and classifications. In the early computers, the processor was composed of one controller unit, and one computation unit (ALU) allowing it to do sequential computations. As the computer use cases evolved, the necessity for increased performance motivated the researchers and processor manufacturers to change the architecture, to enhance the performance. In the next chapter, we follow the evolution of the 8086 processors.

II.10. QCM

1. Which of the following best describes a processor?

A) A device that stores data permanently

B) A component responsible for executing software instructions

C) A memory unit used to cache frequently accessed data

D) An input/output interface for peripherals

2. What is the main goal of modern processor design?

A) Maximize physical size

B) Minimize clock speed

C) Improve performance and efficiency

D) Reduce compatibility with software

3. Which component of a microprocessor is responsible for performing arithmetic and logic operations?

A) Control Unit

B) Registers

C) ALU (Arithmetic Logic Unit)

D) Cache Memory

4. Which of the following is NOT a typical component of a CPU?

A) Program Counter

B) Instruction Register

C) Hard Disk Drive

D) General Purpose Registers

5. What does the Instruction Set Architecture (ISA) define?

A) The layout of motherboard slots

B) The set of registers available to a programmer

C) The electrical voltage levels in the CPU

D) The number of USB ports on the system

6. Which of the following is part of an ISA definition?

A) Physical chip size

B) Supported data types and instruction formats

C) Cooling requirements

D) Number of cores

7. Which addressing mode uses the actual address of the operand in the instruction itself?

A) Immediate addressing

B) Direct addressing

C) Register indirect addressing

D) Indexed addressing

8. In which addressing mode is the operand included directly in the instruction?

A) Indirect addressing

B) Register addressing

C) Immediate addressing

D) Base plus offset addressing

9. What is the first step in the instruction execution cycle?

A) Decode

B) Fetch

C) Execute

D) Write-back

10. During which phase is the instruction interpreted by the control unit?

A) Fetch

B) Execute

C) Decode

D) Memory access

11. What is a key feature of RISC architecture?

A) Large number of complex instructions

B) Variable-length instruction formats

C) Fixed-length instructions and simple decoding

D) Multiple memory operands per instruction

12. Which of the following processors is based on CISC architecture?

A) ARM

B) MIPS

C) x86

D) RISC-V

13. Which memory model allows both code and data to be stored in the same memory space?

A) Harvard architecture

B) Von Neumann architecture

C) RISC architecture

D) CISC architecture

14. Which architecture uses separate memory spaces for instructions and data?

A) Von Neumann

B) Flynn's Model

C) Harvard

D) NUMA

15. According to Flynn's classification, what does SISD stand for?

A) Single Instruction, Single Data

B) Single Instruction, Shared Data

C) Sequential Instructions, Simultaneous Data

D) Streamed Input, Streamed Data

16. Which Flynn category applies to traditional single-core processors?

A) MIMD

B) SIMD

C) MISD

D) SISD

17. Which type of architecture processes multiple instructions on multiple data streams?

A) SIMD

B) SISD

C) MIMD

D) MISD

18. GPUs are often associated with which Flynn classification?

A) SISD

B) MISD

C) SIMD

D) MIMD

19. Which of the following is true regarding RISC vs CISC processors?

A) RISC has more complex instructions than CISC

B) CISC aims to complete tasks in fewer clock cycles

C) RISC typically has a larger number of registers

D) CISC uses fixed-length instructions

20. Which of the following steps is NOT typically part of the instruction cycle?

A) Fetch

B) Translate

C) Decode

D) Execute

II.11. QCM Solution

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
B	C	C	C	B	B	B	C	B	C	C	C	B	C	A	D	C	C	C	B

Chapter 3: Intel 8086 Processor

III.1. Overview of the 80x86 Family

The 80x86 family, developed by Intel, began with the 8086 processor in 1978. It marked the start of the x86 architecture, which became the foundation for most modern personal computers. The 8086 was a 16-bit processor, succeeding the 8-bit 8080. Refer to Table 3.1 for comparison between the 8086 processor and its predecessors in term of characteristics. Table 3.2 compares 8086 and earlier processors 8080 and 8085.

Table 3. 1: Early processors characteristics

Processor	Year	Transistors	Clock rate (MHz)	External data bus	Internal data bus	Address Bus
4004	1971	2250	0,108	4	8	12
8008	1972	3500	0,200	8	8	14
8080	1974	6000	3	8	8	16
8085	1976	6000	6	8	8	16
8086	1978	29000	10	16	16	20

The 80x86 family introduced features like segmented memory addressing, a rich instruction set, and backward compatibility, which allowed newer processors to run software written for older ones.

Table 3. 2: 8080/8085 vs 8086

8086	8080/8085
1 megabyte memory (20-bit address bus)	64 kilobyte memory (16-bit address bus)
16-bit data bus	8-bit data bus
Pipelined processor (1st single chip microprocessor)	Non-pipelined processor

The 80x86 family included other processors which are **the 80286, 303336, and 80186. The 80286** had 16-bit internal and extend data bus and 24-bit address bus. It supported three operations modes, the **virtual memory** based which is a way of access to unlimited memory by swapping data between disk storage and RAM. The **real memory** mode for faster operation, with a maximum of 1 Mbytes of memory. The **protected mode** which is slower but can use 16 Mbytes of memory.

The 80386 processor had 32-bit internal and external data bus as well as 32-bit address bus (2^{32} – 4 gigabyte-physical memory) with a virtual memory of 64 terabytes. 80336SX was later introduced with the same internal structure of 16-bit external data bus and 24-bit address bus. The 80386SX was much cheaper.

The 80486 processor had a 32 internal-external data bus and 32-bit address bus. It was built in math co-processor in a single chip and introduced the use of cache memory which is a static RAM with very fast access time.

III.2. Intel 8086 Architecture

The **Intel 8086** is a 16-bit microprocessor introduced in 1978, which laid the foundation for the x86 family of processors.

III.2.1. 8086 Registers

The **8086 register set** is organized into several groups, each serving a distinct function in computation, memory management, and control flow. All registers are 16 bits wide.

- **General-Purpose Registers:** AX, BX, CX, DX (16-bits)
- **Pointer and Index Registers:** SP (Stack Pointer), BP (Base Pointer), SI (Source Index), DI (Destination Index).
- **Segment Registers:** CS (Code Seg), DS (Data Seg), SS (Stack Seg), ES (Extra Segt).
- **Instruction Pointer (IP):** Holds the offset of the next instruction to execute.
- **Flags Register (FLAGS):** Contains status and control flags (e.g., Carry Flag, Zero Flag, Sign Flag).

III.2.2. Segmented Addressing and Addressing Modes

Segmented Addressing: The 8086 uses a 20-bit address bus but only 16-bit registers. To address more memory, it combines a segment register (16-bit) with an offset (16-bit) to form a 20-bit physical address: Physical Address = (Segment Register << 4) + Offset. **Addressing Modes**: Includes direct, register indirect, based, indexed, and based-indexed addressing.

- **Direct:** Operand's address is given directly, example: MOV A, 1000
- **Register Indirect**: Operand's address is stored in a register, example: MOV A, (R1)
- **Based**: Operand's address is calculated by adding a base register and an offset.
- **Indexed**: Operand's address is calculated by adding an index register and a constant. Example: MOV A, [R2 + 10]
- **Based-Indexed:** Operand's address is calculated by adding a base register, an index register, and a constant, example: **MOV A, [R1 + R2 + 10]**

III.2.3. Instruction Execution Steps in the 8086

Supposing that an instruction is composed of the following sequence of sub-instructions (see Figure 3.1):

Figure 3. 1: Instruction sub-instructions

In **sequential** mode, the instruction needs to be fully executed for another instruction to get loaded. This means that to execute 2 instructions, the processor spends 10-time units, five for each (see the example in figure 3.2).

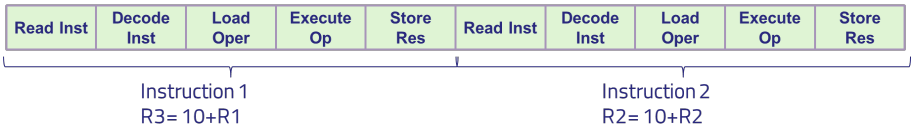

Figure 3. 2: example of sequential execution of instructions

III.3. Instruction Parallelism (Pipelining) - Intel 8086 microprocessor

The 8086 had a **two-stage pipeline** (the prefetch queue and the execution pipeline), which allowed it to fetch the next instruction while the current instruction was being executed. In the **prefetch queue** phase, the 8086 had a 6-byte prefetch queue that stored instructions as they were fetched. This allowed the processor to fetch the next instructions while the current instruction was being decoded and executed. During the **execution** phase once an instruction was in the pipeline, it would be executed in the appropriate stage, and the result would be written back.

III.4. Instruction Parallelism (Pipelining) – Intel 80486 microprocessor

The instruction parallelism introduced in intel 80486 is known as 5 stages pipeline. In **this pipeline** multiple instruction may be executed at the same time in a **shifted** mode known as **Pipeline**, each sub-instruction of which uses uniquely one of the processor components. Therefore, in less than 10-time units (9 units), we are able to execute 5 **pipelined** instructions. **Pipeline** may be used if the instructions have **uniform format** and composed of **sub-instructions** using different parts of the processor (**independent**). Refer to the Figure 3.3 for an example.

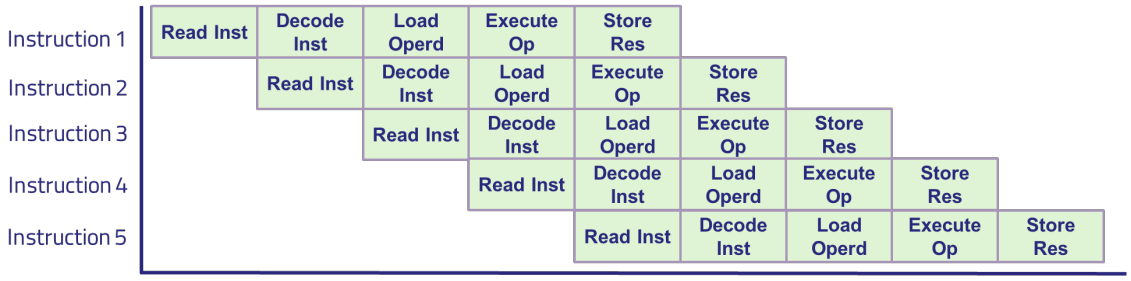

Figure 3. 3: example of pipelined execution of instructions

III.4.1. Pipeline Hazards

As we have mentioned before, the pipelining is possible when the instructions have uniform format, when the data between the instructions is independent and the components are idle. When these conditions are meet, we would have a perfect pipeline like the one illustrated in Figure 3.3. However, the instructions may use the same data resulting in a dependency, the structures may not be idle when needed and be still in use (occupied). Moreover, since we pass 5 instructions to the pipeline, we may load the wrong block on instructions if they are condition dependent. The previously mentioned cases are known as pipeline hazards more precisely data hazards, structural hazards and control hazards respectively. We explain in the following each type of hazard with an example and discuss potential solutions to mitigate them.

- **Structural Hazards**: 2 or more instructions use the **same CPU component** at a given time. (**Solution** is to **halt** the execution of the other instructions until the CPU component becomes **available**)
- **Data Hazards**: When an instruction **depends** on the result of another one. Example: **R1**=10+R3, R2= **R1**+5;
- **Control Hazards**: The "**if instruction**" depend on a value being calculated. The **Branch** to be executed unknown depends on the condition.

III.4.2. Pipeline Data Hazards Solutions

- **Interrupt** the execution of the instruction causing the problem (stall), refer to figure 3.4 for illustration.

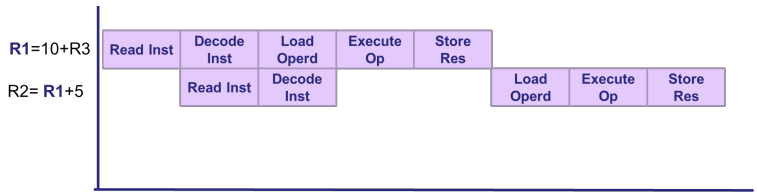

Figure 3. 4: Resolving data hazard by stalling

- **Reorder** the instructions to avoid the issue, refer to figure 3.5 for illustration.

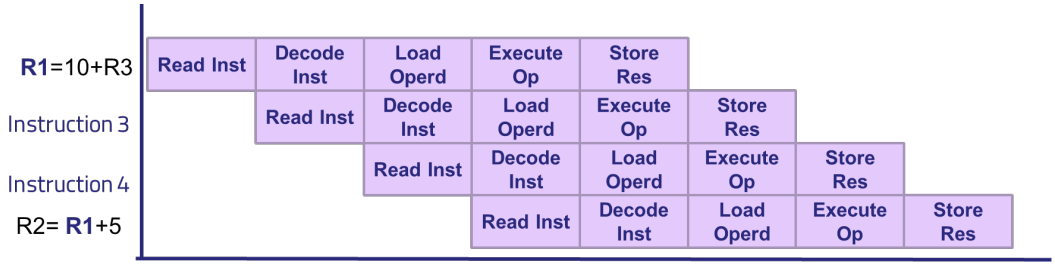

Figure 3. 5: Resolving data hazard by reordering

- **Short-Circuit (Forwarding):** reuse the result from inst1 directly in inst2 without waiting to store in memory and reloading it, refer to figure 3.6 for illustration.

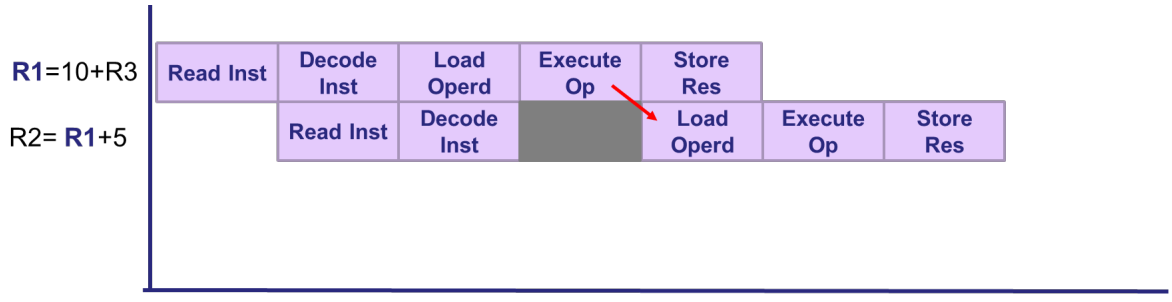

Figure 3. 6: Resolving data hazard by Forwarding

III.4.3. Pipeline Control Hazards Solutions

Example of control hazard:

```
(1) R1=R2*20
```

```
(2) If (R1>30)
    (3) Then R3=10+R1
    (4) Else  R3=20+R1
```

Solution 1: wait until (1) ends to execute (2), depending on it result, either part (3) or (4) is executed.

Solution 2: predict the result of (2) and load either (3) or (4) depending on the prediction result. The prediction algorithm relies on the use of two elements the **Branch Target Buffer (BTB) and Branch History Buffer (BHB)**. The **BTB** contains the addresses of the program branches. The **BHB** saves the branches used before (older choices). To achieve a better prediction, it is advised to increase the size of the BHB and enhance the prediction algorithm quality. **But**, the **large size** may add a **delay**. So does the execution of **good** prediction **Algorithm**.

III.4.4. Execution Time

The pipeline optimizes and reduces the total execution time. To compare between the total execution times, we define their computation equation. $\mathbf{T_e}$ is the time to execute one sub-instruction (cycle).

Total execution time for **m** instructions of **n** sub-instructions in **sequential mode** is:

$$T_t = n * m * T_e$$

Total execution time for **m** instructions of **n** sub-instructions in **pipeline mode** (pipeline of n stages) is:

$$T_t = (n + m - 1) * T_e$$

III.5. High-level and Low-level Programming Languages

ISA aims to provide an efficient execution of high-level programming languages. A simple high-level instruction like: Variable = expression can be translated to a set of low-level calculation instructions {instruction1, …, instruction n} and ends by storing the computed value found in a register to the variable address in memory.

In short, a high-level programming language has:

- **A data Structure**: Variables and Constants. Data can be scalar: integers, characters, reals, or composed like tables and matrixes, records and files. pointers, lists, trees, graphs, etc.
- **Instructions** can be assignments as in: var= exp. **Internal control** within the same program such conditional instructions (if-else) or repetitions (for and while Loops). **External control** known as call of functions and procedures.

A high-level language program is compiled to be executed. The compilation translates the high-level code into an **optimized** low-level program. The optimization is related to the register allocation and usage. See Figure 3.7 for an explanation by example.

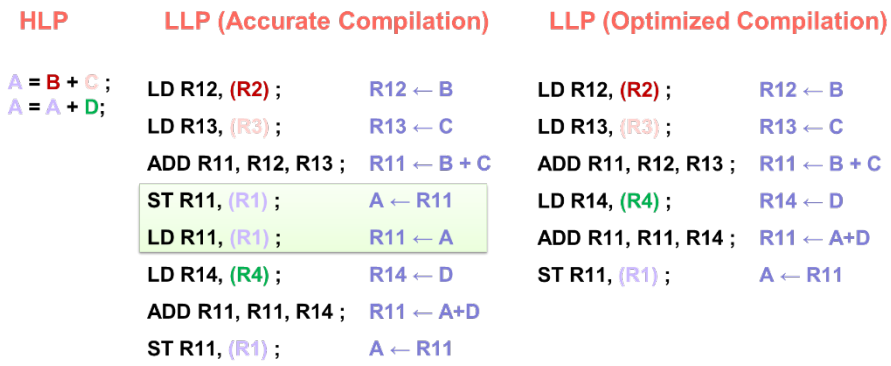

Figure 3. 7: Accurate VS optimized compilation

III.5.1. Netwide Assembler Guide

The **Netwide Assembler (NASM)** is a widely used assembler for the x86 family of processors. Known for its simplicity and portability, NASM supports Intel syntax and is often employed in both academic and practical low-level programming contexts. We resume below a brief guide to Nasm usage.

III.5.1.A.Program structure

A typical NASM program is divided into **sections**, each serving a distinct purpose:

- **.text** – Contains the executable code (instructions).
- **.data** – Holds initialized data or constants.
- **.bss** – Reserves space for uninitialized data.

III.5.1.B.Instruction types

- **Computing Instructions**

These include arithmetic and logical operations directly performed on registers, memory, or immediate values.

- **Arithmetic Operations, few instructions as an example:**

  ADD AX, BX ; AX = AX + BX

  SUB CX, 1 ; CX = CX - 1

  MUL DX ; AX = AX * DX (unsigned)

  DIV BX ; DX:AX / BX ⟶ Quotient in **AX**, Remainder in **DX**

- **Logical Operations**

```
AND AX, 0F0Fh  ; Bitwise AND
OR BX, 0001h   ; Bitwise OR
XOR DX, DX     ; Clear register (DX = 0)
NOT AL         ; Bitwise NOT
```

- **Shift and Rotate**

```
SHL AX, 1      ; Shift left (multiply by 2)
SHR AX, 1      ; Shift right (unsigned divide by 2)
ROL AL, 1      ; Rotate bits left
ROR AL, 1      ; Rotate bits right
```

These instructions are essential for low-level arithmetic, flag manipulation, and bitwise control, especially in hardware or performance-critical contexts.

- **Control Instructions**

Control instructions alter the flow of execution by modifying the instruction pointer based on conditions or jumps.

- **Unconditional Jump**

```
JMP target_label  ; Jump to label unconditionally
```

- **Conditional Jumps (after a CMP instruction)**

```
CMP AX, BX        ; Compare AX and BX
JE equal_label    ; Jump if equal
JNE not_equal     ; Jump if not equal
JL  less_label    ; Jump if less (signed)
JG  greater_label ; Jump if greater (signed)
```

- **Return & Interrupts**

```
RET           ; Return from procedure
INT 21h       ; Software interrupt (e.g., DOS services)
```

- **Loop and While Constructs**

Although NASM has a LOOP instruction, it lacks high-level loop constructs found in languages like C. Loops are generally implemented using LOOP or through structured use of labels and jump instructions.

- **Counter-Based Loop (LOOP instruction)**

The LOOP instruction decrements CX and jumps to a label if CX ≠ 0.

```
MOV CX, 5
loop_start:
; loop body
```

```
LOOP loop_start
```

- **- WHILE-like Loop (Manual Implementation)**

A while loop can be constructed using a label and conditional jump:

```
MOV AX, 0
while_start:
CMP AX, 10
JGE while_end    ; Exit loop if AX <= 10
; loop body
INC AX
JMP while_start
while_end:
```

Note: Comment in NASM starts by ";".

III.5.1.C.Data types

NASM works with low-level data units:

- BYTE (8 bits)
- WORD (16 bits)
- DWORD (32 bits)
- QWORD (64 bits, in x86-64 mode)

To define a variable of those types, refer to Table 3.3 for both cases initialized and uninitialized variables.

Table 3. 3: Defining initialized and uninitialized variables

Initialized variable		**Uninitialized variable**	
section .data a DB 0x1F	; 8-bit variable initialized to 31 (hex)	section .bss buffer RESB 64	; Reserve 64 bytes
b DW 1000	; 16-bit variable initialized to 1000	counter RESW 1	; Reserve 1 word (2 bytes)
c DD 120	; 32-bit variable initialized to 120	V2 RESD 1	; Reserve 1 doubleword
d DQ 10	; 64-bit variable initialized to 10	V3 RESQ 1	; Reserve 1 quadword
T DW 10, 20, 30	; Array of 16-bit integers	Array RESD 100	; Reserve 100 doublewords (400 bytes)

Nasm uses registers with 8, 16,32, and 64 bits. There naming is shared bellow; the first 8 registers have double name.

- **64-bits Registers:** R0(RAX), R1 (RCX), R2 (RDX), R3 (RBX), R4 (RSP), R5 (RBP), R6 (RSI), R7(RDI), R8, R9, R10, R11, R12, R13, R14, R15
- **32-bits Registers:** R0D(EAX), R1D(ECX), R2D (EDX), R3D(EBX), R4D(ESP), R5D(EBP), R6D(ESI), R7D(EDI), R8D, R9D, R10D, R11D, R12D, R13D, R14D, R15D

- **16-bits Registers:** R0W(AX), R1W(CX), R2W (DX), R3W(BX), R4W(SP), R5W(BP), R6W(SI), R7W(DI), R8W, R9W, R10W, R11W, R12W, R13W, R14W, R15W
- **8-bits Registers:** R0B(AL), R1B(CL), R2B (DL), R3B(BL), R4B(SPL), R5B(BPL), R6B(SIL), R7B(DIL), R8B, R9B, R10B, R11B, R12B, R13B, R14B, R15B

III.5.2. Assembly Language use cases

While often associated with legacy systems, assembly language continues to serve specific roles where direct hardware control, execution speed, or minimal memory usage is required. Common use cases include:

- **Reverse Engineering:** Disassembling binaries to analyze behavior or recover algorithms, especially when source code is unavailable.
- **Security:** Writing exploits, analyzing malware, or inspecting low-level vulnerabilities.
- **Hardware Manipulation:** Direct interaction with device registers and memory-mapped components, particularly in system boot routines.
- **Routines (drivers):** Writing tightly controlled code for hardware drivers or critical routines where latency and timing are crucial
- **OS-dependent tasks:** Implementing context switching, interrupt handling, or low-level memory management.
- **Complex Calculations:** Optimizing complex calculations in performance-sensitive applications such as graphics, simulation, or cryptography.
- **Embedded Systems:** Programming microcontrollers and low-resource devices, where efficiency and deterministic behavior are essential.

III.5.3. Debugging in Linux

Linux must have both NASM & GCC installed, the debugging process enables you to see the registers value as you are executing the program step by step. To debug a NASM program, follow these steps:

1- assemble and execute the program

```
nasm -f elf64 -g -F dwarf -o example.o example.asm
```

or

```
nasm -f elf32 -g -F dwarf -o example.o example.asm
```

ld -o example example.o

or

ld -m elf_i386 -s -o example example.o

2- debug the program

gdb exemple

3- specify the breakpoint : _start symbol

b _start

4- execute the program until the breakpoint _start

run

5- execute the program line by line

s

6- see the content of registers:

info registers or i r

7- see the content of a specific register

i r <register>

8- type q to quite the gdb

III.6. I/O Management

A computer can exchange information with its environment; these exchanges of information are called inputs/outputs (or IO).

- **Input**: it is the flow of information coming to the computer from the outside through keyboard, network, hard disk, etc.
- **Output**: it is the flow of information emitted by the computer to the outside through display screen, network, hard disk, etc.

During an input/output operation, information is exchanged between the main memory and a peripheral connected to the computer. This exchange requires an interface or controller (electronic circuit managing the connection). The interface performs more or less complex functions depending on the type of device. Its main purpose is to unload the processor to improve system performance.

III.6.1. Computers Buses

The buses connecting the processor to the main memory (local buses) are the address bus, the data bus and the command bus (R/W type command signals). The local bus is the fastest bus. The local bus is also connected to extension bus controllers, and sometimes to cache memory controllers. Extension buses (or input/output buses) allow extension controllers (cards) to be connected to the PC using special connectors (slots on the motherboard). Extension controllers are used to link the PC to input/output peripherals.

III.6.1.A. Local Bus and extension Bus of ISA type

III.6.1.B.ISA Bus

The ISA (Industry Standard Architecture) expansion bus depicted in figure 3.8 is the most common on PCs. With a relatively low frequency and weak characteristics, it is used to connect relatively slow cards (modems, sound cards, etc.). The main characteristics of the ISA bus are: 16 data bits, 24 address bits, 16 interruption lines, 8 MHz frequency. One of the first types of expansion bus was the ISA, which was used with IBM-compatible computers. However, the ISA bus had a tendency to bottleneck and was replaced by the PCI bus.

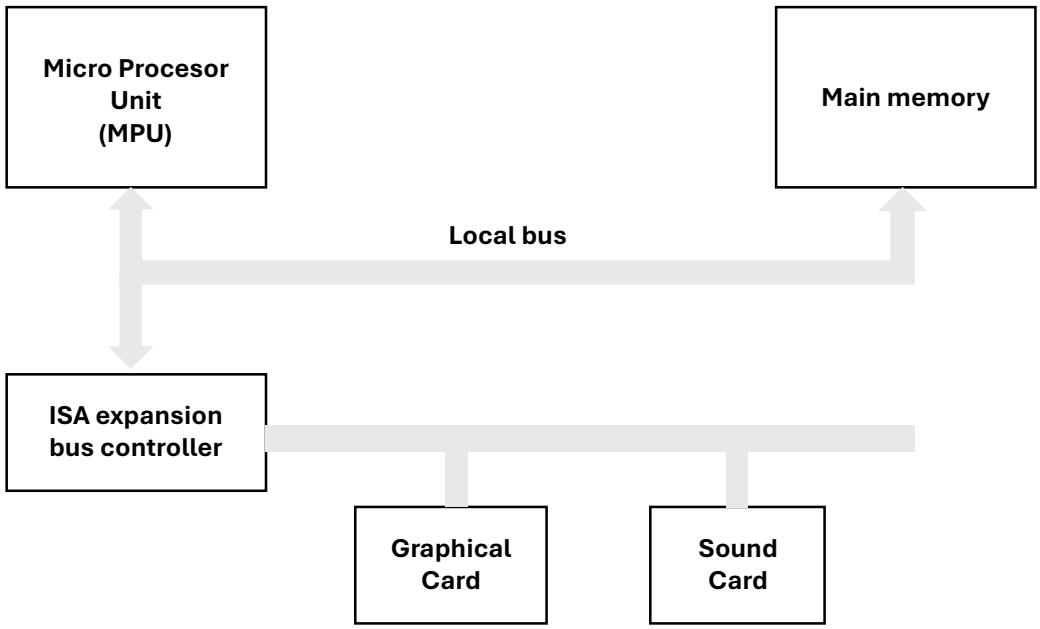

Figure 3. 8: local and extension ISA buses

III.6.1.C. PCI local bus

"Modern" input/output peripherals require very large information transfers between the main memory and the controller. For example, a recent SVGA graphics card has a video memory of 1 to 8 MB, and implements transfers between this memory and the MP at 60 MB/s.

To allow such speeds, it is necessary to connect the peripheral controller directly to the local bus. The controller thus benefits from the high speed of this bus; moreover, it can take control of it to carry out transfers directly with the MP without going through the processor.

The PCI bus (Peripheral Component Interface) shown in figure 3.9 equips the vast majority of recent PCs. The principle of the PCI bus is precisely to dissociate the processor and the buses.

This separation makes it possible to use a bus frequency different from that of the processor and facilitates the evolution of machines. The characteristics of the PCI bus are: 32 or 64 data bits, 32 address bits, 33 MHz frequency. It allows speeds of 132 MB/s in 32 bits, or 264 MB/s in 64 bits.

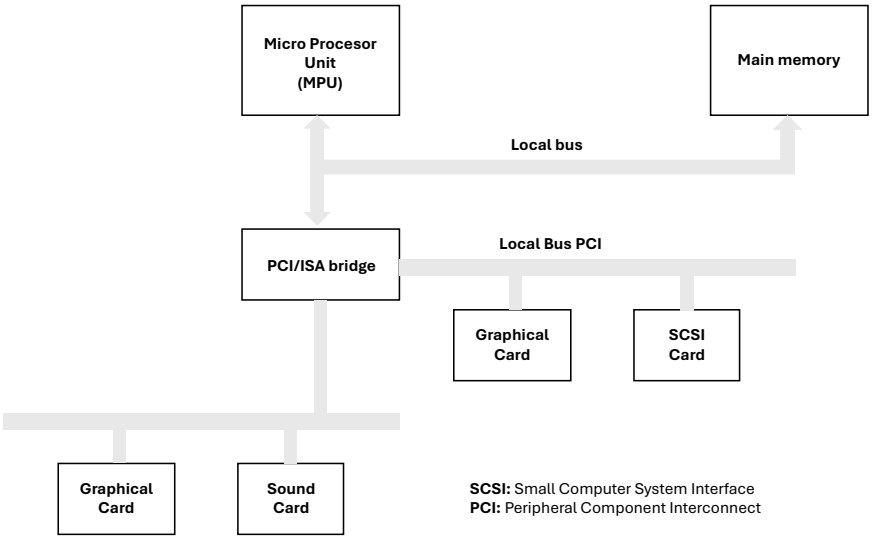

Figure 3. 9: PCI Bus

III.6.1.D. Peripheral Buses

These buses make it possible to connect an interface (controller) of the computer to one or more peripherals (generally outside the computer). The SCSI bus (Small Computer System Interface) is a parallel input/output bus which allows the connection of 1 to 7 peripherals of all kinds (hard disks, CD/ROM readers, digitizers (scanners), tape readers (streamers) , ... SCSI version 1 allows a transfer rate of 4 MB/s (8 bits wide). SCSI version 2 provides up to 40 MB/s in 32 bits.

The PCMCIA (Personal Computer Memory Card International Association) bus is an extension bus used on Laptops. The data exchanged between a peripheral and the processor passes through the interface (or controller) associated with this peripheral. The interface has a buffer memory to store the data exchanged (depending on the type of interface, this buffer memory ranges from 1 single byte to a few megabytes).

The interface also stores information to manage communication with the device:

- **command information**, to define the operating mode of the interface: transfer direction (input or output), data transfer mode (by scrutiny or interrupt), etc. This command information is communicated to the interface during its initialization phase, before the start of the transfer.
- **status information**, which memorizes the way in which the transfer was carried out (transmission error, reception of information, etc.). This information is intended for the processor.

Data from each interface is accessed through an I/O address space, which is accessed by the IN and OUT instructions of the 80x86.

- IN AL, I/O address: reads the specified address byte from the input/output space and transfers it to the AL register.
- OUT I/O address, AL: writes the contents of AL to the specified address of the input/output space.

During the execution of the IN and OUT instructions, the processor sets its IO/M terminal to 1 and presents the I/O address on the address bus. The IO/M signal indicates to the address decoding circuits that it is not a main memory address, but the address of an input/output interface.

III.6.2. Transfer Modes

Data transfer between the processor and the interface can be done in different ways. A distinction is made between unconditional transfers and conditional transfers to the device. Unconditional transfers are the simplest; they concern very simple peripherals (interrupter, indicator lights, ...) which do not have a status register and are always ready. Transfers with condition are more complex: before sending or receiving information, the processor must know the state of the peripheral (for example, in reception on a network link, one must know if a byte has arrived before requesting the reading this byte). There are two methods here, transfer through polling (active waiting) and interruption.

III.6.3. The asynchronous serial input/output interface

The serial input/output interface equips all PCs and allows the exchange of information at low speed with a peripheral such as a modem, or with another PC, over distances of less than a few tens of meters. The electronic component responsible for managing asynchronous serial transmissions in PCs is called UART (Universal Asynchronous Receiver Transmitter). Figure 3.10 how a byte is transferred between two computers, note that the transmission of a byte b7b6b5b4b3b2b1b0 in serial starts from the least significant to the most significant.

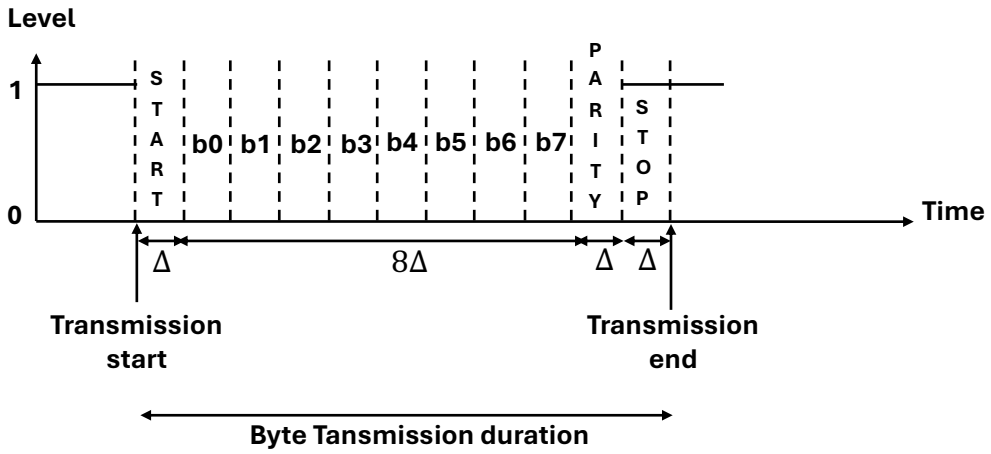

Figure 3. 10: byte transmission

III.6.4. Different transmission types used to connect two PCs

When connecting two personal computers for the purpose of data exchange, several transmission methods can be employed depending on the hardware interfaces, required speed, and physical distance. These methods are categorized based on how data is transferred—serially or in parallel—and whether the connection is direct or through a network medium. Refer to Figure 3.11 for parallel transmission illustration, Figure 3.12 for serial transmission.

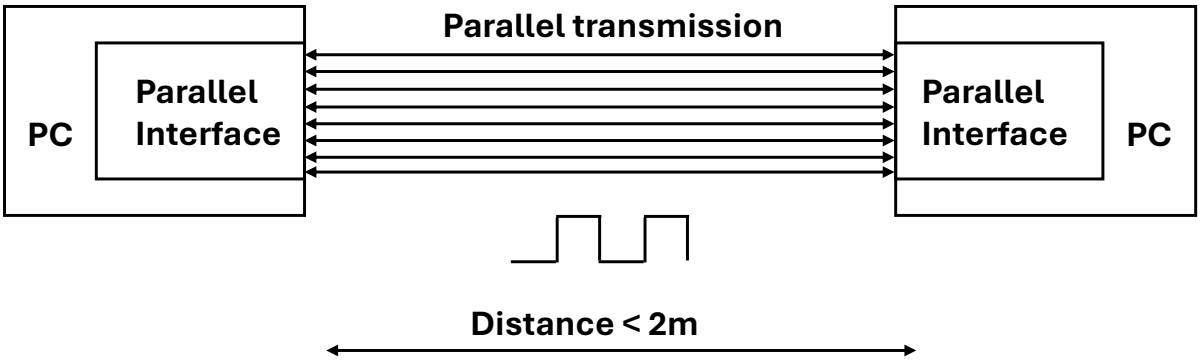

Figure 3. 11: Parallel Transmission

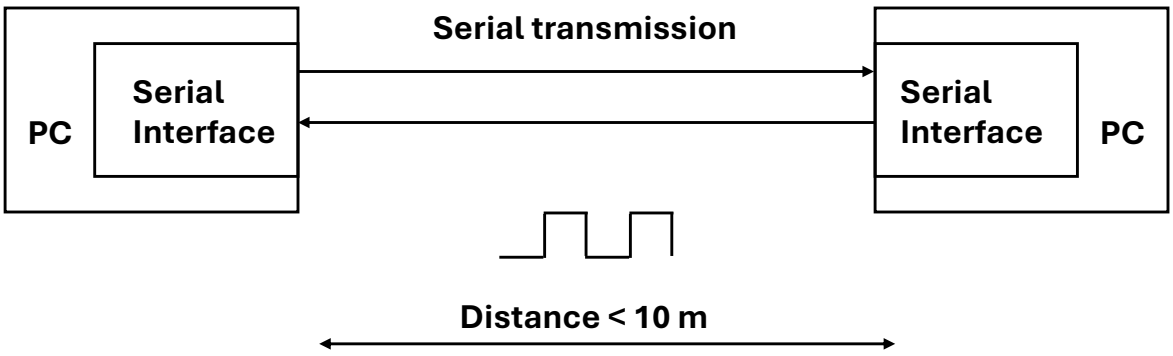

Figure 3. 12: Serial Transmission

III.7. Stack managements

The stack is a fundamental structure in low-level system architecture, providing temporary storage for data during program execution. In the 8086 microprocessors, the stack is managed explicitly through dedicated instructions and plays a critical role in function calls, parameter passing, local variable storage, and interrupt handling.

III.7.1. STACKS

The stack is a list of data words. It uses the Last in First Out (LIFO) access method which is the most popular access method in most of the CPU. Stack can save multiple values at the same time. Operation type instruction does not need the address field. A register is used to store the address of the topmost element of the stack which is known as Stack pointer (SP).

- **SS (Stack Segment):** contains the address of the stack segment, it is initialized at the beginning of the program is does not change its value.
- **SP (Stack Pointer):** the register that points to the top of the stack.

III.7.2. Instructions PUSH & POP

The instructions **PUSH** and **POP** are fundamental to managing the stack in the 8086 architectures.

- **PUSH register:** pushes the content of a register into the stack (Refer to Figure 3.13 for illustration)
- **POP register:** pops the last inserted value into a register (Refer to Figure 3.13 for illustration).

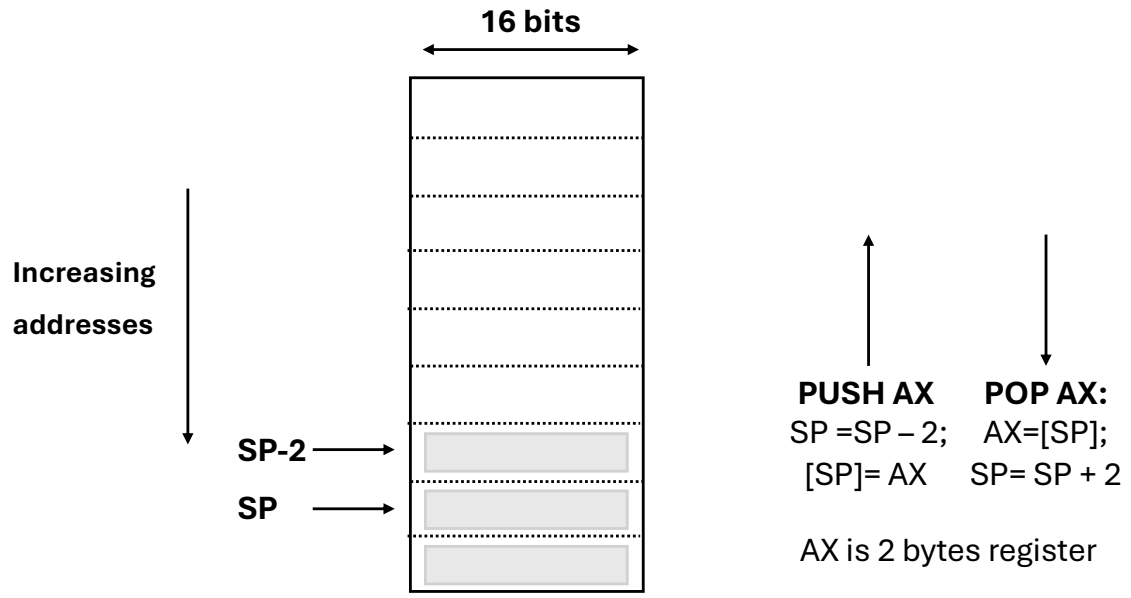

Figure 3. 13: Push and Pop Functioning

III.8. Functions and subroutines

In assembly language, functions—also referred to as subroutines—are reusable blocks of code that perform specific tasks. Unlike in high-level languages, subroutine handling in assembly requires explicit control over calling, returning, and parameter management.

III.8.1. Function call

A Function call is branch to an address. The address is that of the function code loaded in the memory. When calling a function, the address of the instruction following the call is saved. So, a function has a call address and return address. (Refer to Figure 3.14 for illustration)

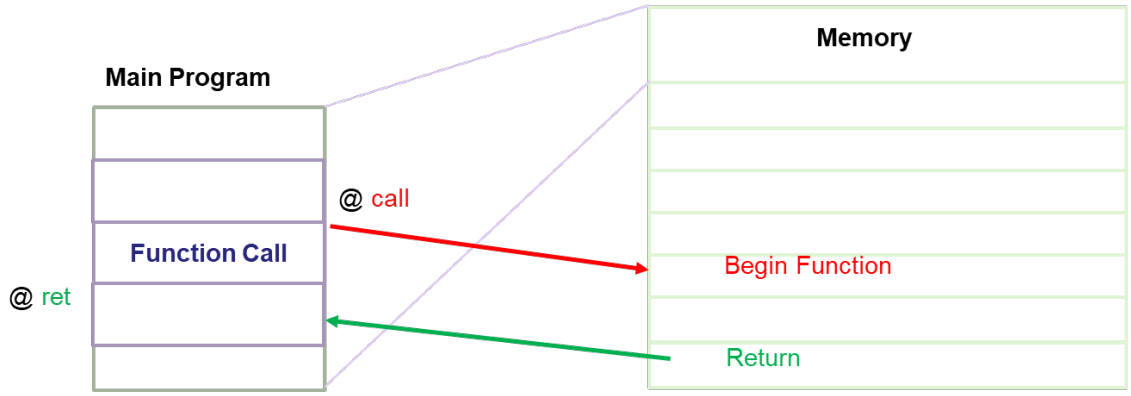

Figure 3. 14: Function call explained

- A Function call

- Push @Ret into Stack
- Jmp @function (IP=@Function)

- Return (Ret)
  - Pop Stack (IP= SP)

III.8.2. Passing out the parameters to the function

In assembly language, **passing parameters to a function** (also referred to as a subroutine or procedure) requires explicit handling, as there are no implicit calling conventions like in high-level languages. The most common methods for parameter passing in x86 assembly are via **registers**, the **stack**, or a combination of both. Refer to Table 3.4 for illustration by example.

Table 3. 4: Passing out parameters to the functions

Example:	**Using Registers**	**Using stack without pulling the parameters**

Sum_a (a , b: int):int
Begin
Return a=a + b
End
	Mov AX, a
	Mov BX, b
	Call Sum_a
	Mov AX, CX
	Sum_a: Mov CX, AX
	Add CX, BX
	Ret
		Push a
		Push b
		Call Sum_a
		Sum_a: Push BP
		Mov BP, SP
		Mov AX, [BP+16]
		ADD AX, [BP+12]
		Pop BP
		ret

III.9. Interrupts

An **interrupt** is a mechanism by which the normal flow of program execution is temporarily halted in order to respond to an event requiring immediate attention. In the context of the 8086 microprocessors, interrupts play a critical role in enabling responsive and efficient handling of hardware signals, system calls, and exceptions.

III.9.1. Definition

Interruptions allow hardware to communicate with the processor. When you want the processor to react quickly to an external event, such as: arrival of a data packet on a network connection, typing of a character on the keyboard, modification of the time. Interruptions are mainly used for managing computer peripherals. An interruption is signaled to the processor by an electrical signal on a special terminal. When this signal is received, the processor "processes" the interrupt as soon as the instruction it was executing ends.

Interruption processing consists of either:

- To ignore it and proceed normally to the next instruction: this is possible only for certain interrupts, called maskable interrupts. It is indeed sometimes necessary to be able to ignore the interruptions for a certain time, to carry out urgent processing for example. When the processing is finished, the processor unmasks the interrupts and then takes them into account.
- To execute an Interrupt Handler. It is a program that is called automatically when an interruption occurs. The beginning address of the handler is given by the interrupt vector table. Upon the end of the interruption code execution, it executes the special instruction IRET which resumes the execution of the program that was interrupted.

III.9.2. Hardware interruptions on PC

In a PC system based on the x86 architecture, hardware interrupts are essential for communication between peripheral devices and the CPU. These interrupts allow external components—such as keyboards, timers, disk controllers, and network cards—to notify the processor of events requiring immediate processing. The 8086 microprocessor handles hardware interrupts through a dedicated interrupt controller, known as Programmable Interrupt Controller (PIC), which manages and prioritizes multiple interrupt requests before forwarding them to the CPU.

III.9.2.A. Interrupt signals

80x86 family processors have three terminals for handling interrupts: NMI, INTR, and INTA.

- **NMI (NonMaskable Interrupt)** is used to send a non-maskable interruption to the processor. The processor cannot ignore this signal and will execute the processing given by the interruption vector. This signal is normally used to detect hardware errors (such as faulty main memory).
- **INTR (Interrupt Request),** maskable interruption request. Used to indicate to the processor the arrival of an interruption.
- **INTA (Interrupt Acknowledge)** This pin is set to 0 when the processor actually processes the interruption signaled by INTR (i.e. it is no longer masked)

III.9.2.B.Indicator IF

At any given time, interrupts are either masked (ignored) or enabled (authorized), depending on the state of a special flag in the status register, IF (Interrupt Flag). The state of IF indicator may be changed through two instructions, CLI (Clear IF, sets IF to 0), STI (SeT IF, sets IF to 1).

- **if (IF = 0)**, the Processor **ignores** the interruption
- **if (IF = 1)**, the processor accepts the interruption request (maskable), i.e. it is processed immediately.

III.9.2.C. Interruption Controller

The computer is connected to several peripherals, but we have just seen that there was only one interrupt request signal, INTR. The interruption controller is a special circuit, external to the processor, whose role is to distribute and queue the interrupt requests coming from the various peripherals. The controller is connected to the interfaces managing the peripherals by the IRQ terminals. It manages the interrupt requests sent by the peripherals, to send them one by one to the processor (via INTR). Before sending the next interruption signal, the controller waits to have received the signal $\overline{INTA}$, indicating that the processor has successfully processed the current interruption. Refer to Figure 3.15 for illustration.

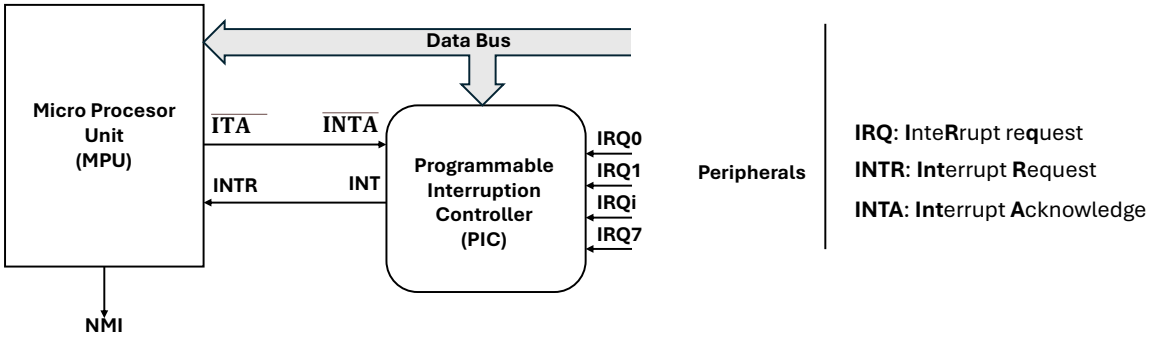

Figure 3. 15: maskable and non-maskable interruptions controller

An INT signal is emitted by a peripheral. The interrupt controller receives this signal on one of its terminals IRQi. As soon as possible, the controller sends a signal to its INT terminal. The MPU takes into account the signal on its INTR terminal after having completed the execution of the current instruction. If the flag IF=0, the signal is ignored, otherwise the interrupt request is accepted. If the request is accepted, the MPU sets its INTA output to level 0 for 2 clock cycles, to indicate to the controller that it is taking its request into account. In response, the interruption controller places the interruption number associated with the IRQi terminal on the data bus. The processor reads the interruption number from the data bus and uses it to find the interruption vector. The procedure handling the interruption is executed. During this time, interruptions are masked (IF=0). If the processing is long, it is possible in some cases to re-authorize the interrupts with the STI instruction.

III.9.3. Software Interruption (System calls)

The INT n instruction calls the nth function of the interruption vector table. n is an integer between 0 and 255 (1 byte), because there are 256 interrupt vectors in the table. The INT n instruction is similar to the CALL instruction.

The execution of INT n follows these steps:

1. saves the flags of the status register on the stack (the flags are grouped in a 16-bit word);

2. pushes current CS & IP in the stack; (CS: code segment register)

3. CS & IP load the value read at address n, where n is the parameter of INT.

4. Therefore, the execution continues at the beginning of the interruption handler.

III.10. Conclusion

In this chapter, we explored the **Intel 8086 family of processors**, focusing on their architectural features and foundational concepts. A key topic discussed was the **segmented memory model**, which defines how memory is organized and accessed through various **addressing modes**. This structure was essential in the early design of x86 processors, allowing them to manage larger address spaces than would otherwise be possible with a 16-bit address bus.

The chapter also introduced the concept of **instruction pipelining**, a technique used to enhance processor performance by overlapping instruction execution stages. We examined how pipelining allows for improved instruction throughput and explained the concept of **pipeline hazards**—including structural, data, and control hazards—that can disrupt the execution flow. Corresponding strategies for hazard resolution, such as instruction reordering and pipeline stalling, were also covered.

Following this, we presented an overview of **assembly language programming using NASM** (Netwide Assembler) for the 8086 architectures. Key syntax and instruction formats were discussed to demonstrate how low-level programming directly interfaces with hardware.

Additionally, the chapter examined **input/output (I/O) management**, including techniques for interfacing with external devices using programmed I/O and interrupt-driven methods. We also reviewed the principles of **stack management**, emphasizing its role in function calls, parameter passing, and local variable storage. Finally, the mechanism and handling of **hardware and software interrupts** were explored, showing how they facilitate efficient context switching and asynchronous event handling within the system.

Overall, this chapter laid the groundwork for understanding the operational principles of early x86 processors, while providing practical insight into low-level programming and system control mechanisms.

III.11. QCM

1. Which processor introduced protected mode?

a) Intel Pentium

b) Intel 8086

c) Intel 80286

d) Intel 80386

2. What type of microprocessor is the Intel 8086?

a) 4-bit

b) 32-bit

c) 8-bit

d) 16-bit

3. Which of these is the first x86 family member?

a) Intel 80386

b) Intel 80186

c) Intel 80286

d) Intel 8086

4. How many general-purpose registers does the 8086 have?

a) 6

b) 10

c) 8

d) 4

5. Which register serves as the instruction pointer in the 8086?

a) BP

b) SP

c) IP

d) SI

6. Which segment register is associated with the stack in 8086?

a) DS

b) CS

c) SS

d) ES

7. What is the main benefit of pipelining in the 8086?

a) Reduces power consumption

b) Increases memory size

c) Simplifies programming

d) Speeds up execution by overlapping fetch and execute

8. How many bytes can the 8086 instruction queue hold?

a) 2

b) 8

c) 4

d) 6

9. What kind of pipelining does the 8086 use?

a) Branch prediction

b) Superscalar

c) Out-of-order

d) Instruction prefetch queue

10. Which language is closest to machine code and specific to a processor's architecture?

a) C

b) Assembly

c) Java

d) Python

11. Which of the following is NOT considered a high-level language?

a) COBOL

b) FORTRAN

c) BASIC

d) Machine Language

12. Which instruction is used to read from an I/O port in 8086?

a) MOV

b) IN

c) OUT

d) XCHG

13. In 8086, how many I/O ports can be addressed directly using IN and OUT instructions?

a) 256 K

b) 64 K

c) 512 K

d) 128 K

14. Which register keeps track of the top of the stack in 8086?

a) BP

b) IP

c) SP

d) SI

15. Which segment register works with SP to locate the stack in memory?

a) CS

b) DS

c) ES

d) SS

16. Which instruction returns control from a subroutine back to the calling program?

a) INT

b) CALL

c) JMP

d) RET

17. Which instruction is used to call a subroutine in 8086 assembly?

a) RET

b) JMP

c) INT

d) CALL

18. Which interrupt is triggered when a division by zero occurs in 8086?

a) INT 3

b) INT 1

c) INT 0

d) INT 4

19. Which of the following is a software-generated interrupt in 8086?

a) NMI

b) RESET

c) INTR

d) INT 21h

20. Which type of interrupt has the highest priority in 8086?

a) Maskable hardware interrupts (INTR)

b) Software interrupts

c) None of the above

d) Non-Maskable Interrupt (NMI)

III.12. QCM Solutions

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
C	D	D	C	C	C	D	D	D	B	D	B	B	C	D	D	D	C	D	D

Chapter 4 : 32-bit and 64-bit Processors

IV.1. Introduction

As the processors continued to evolve, the need for larger buses and registers was mandatory to benefit from the powerful processing speed and avoid the bottleneck issue. As we have seen in the previous chapters, the initial processors started by 4-bits and progressively expanded to 8-bits, then to 16-bits moving forward to 32, 64, and 128-bits for registers and buses sizes. Naturally, the word memory followed as well. In this chapter, we highlight the key differences between the 32-bits and 64-bits processors and the changes that they brought the computer architecture in overall.

IV.2. 32-bits VS 64 bits Main differences:

The evolution from 32-bit to 64-bit architectures represents a significant leap in computing power, memory addressing, and software capability. Here are the key distinctions:

- **Memory limitations and capabilities:**

The 32-bit processors can address up to 4 GB of RAM (2^{32} bytes), while the 64-bit processors can theoretically address up to 16 exabytes (2^{64} bytes), though practical limits are much lower due to hardware and OS constraints.

- **Difference in Performance:**

The 64-bit processors can handle larger data chunks per clock cycle, improving performance for applications that require high precision or large data sets. The 64-bit architectures often have more registers, which can reduce the number of memory accesses and speed up execution.

- **Compatibility concerns:**

32-bit processors can only run 32-bit operating systems and applications. 64-bit processors can run both 32-bit and 64-bit operating systems and applications, though 32-bit applications run in a compatibility mode.

IV.3. Evolution of Architectures

Understanding the evolution of processor architectures gives context to the jump from 32 to 64 bits.

- **IA-32 (Intel Architecture 32-bit): Introduced** by Intel with the 80386 processor, it became the standard for 32-bit computing.
- **AMD64 (x86-64):** Developed by AMD, this architecture extended the 32-bit IA-32 to 64-bit, maintaining backward compatibility. Intel later adopted this architecture as Intel 64.

IV.4. Registers in 32-bit and 64-bit Processors

Registers are small memory locations inside the CPU used to hold temporary data during execution. The differences in registers between 32-bit and 64-bit architectures are both quantitative and structural.

- **General-Purpose Registers:**

32-bit: 8 general-purpose registers (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP).

64-bit: Extends these to 64-bit (RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP) and adds 8 new general-purpose registers (R8-R15).

- **Pointers and Segment Registers:**

32-bit: Uses segment registers (CS, DS, SS, ES, FS, GS) for memory segmentation.

64-bit: Moves to a flat memory model, reducing the reliance on segment registers. The CS, DS, ES, SS registers are still present but largely ignored in 64-bit mode.

- **Extensions of Registers in 64-bit Processors:**

64-bit registers allow for larger data types and more efficient processing of large data sets.

RIP (Instruction Pointer): Extended to 64 bits, allowing for a larger address space for instructions.

IV.5. Advanced Addressing Modes in 32-bit and 64-bit Processors

The method by which a CPU accesses memory—known as addressing—has also evolved between 32-bit and 64-bit designs. While the basic concepts remain the same, 64-bit architecture supports more complex and flexible forms of addressing.

- **32-bit:** Uses segmented addressing, where memory is divided into segments (code, data, stack, etc.), each with its own base address.
- **64-bit:** Uses flat addressing, where the entire memory space is linear and contiguous, simplifying memory management.
- **Base + Offset:**
    - **32-bit:** MOV EAX, [EBX + 4] (loads the value at the address EBX + 4 into EAX).
    - **64-bit:** MOV RAX, [RBX + 8] (loads the value at the address RBX + 8 into RAX).
- **Indexed Addressing:**

- **32-bit:** MOV EAX, [ESI + EDI*4] (loads the value at the address ESI + EDI*4 into EAX).
- **64-bit:** MOV RAX, [RSI + RDI*8] (loads the value at the address RSI + RDI*8 into RAX).

- **Relative Addressing:**
  - **32-bit:** JMP +0x10 (Jump 16 bytes ahead of the current EIP).
  - **64-bit:** JMP +0x10 (Jump 16 bytes ahead of the current RIP).

IV.6. Instruction Set in 32-bit and 64-bit Processors

The basic **instruction** set in 64-bit processors builds on the legacy 32-bit set but adds support for more powerful operations, especially those useful in data-heavy tasks.

- **SSE (Streaming SIMD Extensions):** Introduced with Pentium III, allows for parallel processing of multiple data points.
- **AVX (Advanced Vector Extensions):** Further extends SIMD capabilities, allowing for 256-bit and 512-bit wide registers.

Comparative examples:

- **32-bit ADD Instruction:** ADD EAX, EBX **; EAX = EAX + EBX**
- **64-bit ADD Instruction:** ADD RAX, RBX **; RAX = RAX + RBX**
- **SSE Example:** MOVAPS XMM0, XMM1 **; Move aligned packed single-precision values from XMM1 to XMM0**
- **AVX Example:** VADDPS YMM0, YMM1, YMM2 **; Add packed single-precision values in YMM1 and YMM2, store result in YMM0**

IV.7. Instruction Execution Cycle in 32-bit and 64-bit Processors

Despite the many architectural differences, the fundamental steps a CPU takes to execute instructions remain largely the same in both 32-bit and 64-bit systems. These steps make up what is known as the instruction cycle.

1. **Fetch**: The instruction is fetched from memory.
2. **Decode**: The instruction is decoded into control signals.
3. **Execute**: The operation is performed (e.g., addition, subtraction).
4. **Memory Access**: If the instruction involves memory, the data is read from or written to memory.

5. **Write Back**: The result is written back to the destination register.

- **Example: MOV Instruction in 64-bit: MOV RAX, RBX ;** Move the contents of RBX into RAX

  Fetch: The MOV RAX, RBX instruction is fetched from memory.

  Decode: The CPU decodes the instruction to understand it needs to move data from RBX to RAX.

  Execute: The data in RBX is copied to RAX.

  Memory Access: Not required for this instruction.

  Write Back: The result (data from RBX) is written to RAX.

- **Example: ADD Instruction in 64-bit: ADD RAX, RBX** ; RAX = RAX + RBX

  Fetch: The ADD RAX, RBX instruction is fetched.

  Decode: The CPU decodes the instruction to understand it needs to add RBX to RAX.

  Execute: The addition operation is performed.

  Memory Access: Not required for this instruction.

  Write Back: The result (RAX + RBX) is written to RAX.

- **Example: JMP Instruction in 64-bit: JMP 0x00400000** ; Jump to the address 0x00400000

  Fetch: The JMP instruction is fetched.

  Decode: The CPU decodes the instruction to understand it needs to jump to a new address.

  Execute: The program counter (RIP) is updated to the new address.

  Memory Access: Not required for this instruction.

  Write Back: Not required for this instruction.

IV.8. Conclusion

To recap, we summarize the key differences between the 32-bit processor and the 64-bit processor in Table 4.1.

Table 4. 1: Recap of the main differences

Feature	**32-bit Architecture**	**64-bit Architecture**
Memory Capacity	Up to 4 GB (2^{32} bytes)	Up to 16 exabytes (2^{64} bytes) theoretically
Registers	8 general-purpose registers (EAX, EBX, etc.)	16 general-purpose registers (RAX, RBX, R8-R15)
Data Processing	32-bit data chunks per cycle	64-bit data chunks per cycle
Addressing Modes	Segmented memory model	Flat memory model

Compatibility	Runs only 32-bit OS and applications	Runs both 32-bit and 64-bit OS and applications
Instruction Set	Basic instructions (e.g., IA-32)	Extended instructions (e.g., SSE, AVX)
Performance	Limited by smaller registers and memory	Improved due to larger registers and memory
Instruction Pointer	EIP (32-bit)	RIP (64-bit)
Address Space	Limited to 4 GB	Vastly larger (practically limited by hardware)

IV.9. QCM

1. What is the maximum amount of RAM that can be addressed by a 32-bit processor?

a) 2 GB

b) 4 GB

c) 8 GB

d) 16 GB

2. Which of the following best describes a key difference between 32-bit and 64-bit processors?

a) 64-bit supports more USB ports

b) 64-bit allows for larger amounts of RAM

c) 32-bit has faster clock speeds

d) 64-bit uses less power

3. A 64-bit processor can run:

a) Only 64-bit software

b) Only 32-bit software

c) Both 32-bit and 64-bit software

d) Neither 32-bit nor 64-bit software

4. Which architecture extended the x86 instruction set to support 64-bit computing?

a) ARM64

b) IA-64

c) x86-64 (AMD64)

d) MIPS64

5. Which company originally developed the x86-64 architecture?

a) Intel

b) Microsoft

c) AMD

d) IBM

6. Which technology allowed 32-bit processors to access more than 4 GB of memory before 64-bit became common?

a) Hyper-Threading

b) Physical Address Extension (PAE)

c) Virtual Memory

d) Overclocking

7. How wide are general-purpose registers in a 64-bit processor?

a) 8 bits

b) 16 bits

c) 32 bits

d) 64 bits

8. In 64-bit mode, how many general-purpose registers does x86-64 provide?

a) 8

b) 12

c) 16

d) 32

9. Which of the following is NOT a register extension in 64-bit mode?

a) RAX

b) RBX

c) RDI

d) REX

10. Which addressing mode allows accessing data relative to the current instruction pointer in 64-bit processors?

a) Direct addressing

b) Indirect addressing

c) Base-indexed addressing

d) RIP-relative addressing

11. What is the purpose of base-plus-index addressing in 64-bit processors?

a) To increase cache size

b) To manage interrupts

c) To efficiently access array elements

d) To reset the CPU

12. Which instruction set extension introduced wider registers and more registers in 64-bit mode?

a) MMX

b) SSE

c) AVX

d) x86-64

13. Which of the following is true about 64-bit instruction sets compared to 32-bit?

a) They use shorter instructions

b) They have fewer registers

c) They support more complex operations

d) They are backward compatible with 16-bit code only

14. In modern 64-bit CPUs, what technique helps execute multiple instructions per clock cycle?

a) Pipelining

b) Branch prediction

c) Superscalar execution

d) Memory paging

15. What is the typical order of the instruction execution cycle?

a) Execute ⟶ Decode ⟶ Fetch

b) Fetch ⟶ Execute ⟶ Decode

c) Fetch ⟶ Decode ⟶ Execute

d) Decode ⟶ Fetch ⟶ Execute

16. Which of the following is NOT an advantage of 64-bit over 32-bit systems?

a) Larger addressable memory

b) Better performance with large datasets

c) Lower power consumption

d) Ability to run both 32-bit and 64-bit applications

17. Which of the following statements is true?

a) 64-bit OS can run on a 32-bit processor

b) 32-bit OS can fully utilize 8 GB of RAM

c) 64-bit processors can run 16-bit legacy code

d) 32-bit processors are faster than 64-bit processors

18. Why do 64-bit operating systems generally require more disk space than 32-bit versions?

a) They include more drivers

b) They have larger system files and binaries

c) They support more languages

d) They compress data inefficiently

19. Which of the following is a reason to choose a 32-bit OS over a 64-bit one?

a) More available RAM

b) Better performance with old hardware

c) Wider software compatibility

d) Enhanced security features

20. What is a potential disadvantage of using a 64-bit OS on older hardware?

a) It runs faster

b) It may lack driver support

c) It reduces memory usage

d) It improves battery life

IV.10. QCM Solutions

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
B	B	C	C	C	B	D	C	D	D	C	D	C	C	C	C	C	B	B	B

Chapter 05: Modern Processor Architectures

V.1. Introduction

Modern processors have undergone significant transformations over the past few decades, driven by the need for higher performance, energy efficiency, and the ability to handle increasingly complex workloads. Key trends and technological advancements include:

- **Moore's Law and Beyond**: For many years, the number of transistors on a chip doubled approximately every two years, leading to exponential growth in computing power. However, as transistor sizes approach physical limits, the industry has shifted focus towards alternative methods of improving performance, such as multi-core architectures, specialized accelerators, and advanced manufacturing techniques like 3D stacking.
- **Energy Efficiency**: With the rise of mobile computing and data centers, energy efficiency has become a critical concern. Modern processors are designed to deliver high performance while minimizing power consumption, often through dynamic voltage and frequency scaling (DVFS) and advanced power management techniques.
- **Heterogeneous Computing**: Modern processors often integrate different types of processing units, such as CPUs, GPUs, and specialized accelerators (e.g., TPUs for AI workloads), to optimize performance for specific tasks.
- **Instruction Set Architecture (ISA) Innovations**: Newer ISAs, such as RISC-V, and extensions to existing ISAs, like ARM's SVE (Scalable Vector Extension), are being developed to support emerging workloads like machine learning and data analytics.

V.2. Increasing the CPU performance

V.2.1. Frequency

Decreasing CPU cycle time means that the processor needs to be able to execute **more instruction,** this can be achieved by increasing the frequency. However, achieving high frequency is hindered by few setbacks mainly related to the technology used to produce these processors and the one used in making them, the key limits are related to the materials in use, the heating issue and the signal propagation time in the used material, etc. among the proposed solutions is to reduce the fine engraving and to search for potential better technology to replace the semi-conductors use or change the materials used to engrave the circuits.

V.2.2. Cache memory

To reduce the **idle time** spent by the **processor** while **fetching or storing** data/instruction from/in **memory**, the **cache memories** were added within the processors. The processor uses a **selection algorithm** to choose which operand/instruction to put in cache memory and which one to put in the central memory. As time passed and processors evolved, multiple types of cache are used with different levels (L1, L2, L3), types and sizes. Their use made the processor **performance** more **efficient** and **optimized**.

V.2.3. Parallelism

Executing instructions in sequential mode due to single units of different types was limiting the performance of the processor, as such, researchers aiming to increase its performance thought of duplicating the calculation units. However, they had to think of the adjustments to make in this case. One of the key issues is the number of units to add and how to synchronize between these units, the changes to make to the memory, the buses etc. When considering parallelism, vectorial units were the first to include, then duplicating units, cores then processors to achieve the highest performance through parallelism.

- **Vectorial Calculation:** SIMD, executing the same instruction on a set of data. Refer to Figure 5.1 to see the difference between scalar and vectorial calculation units.

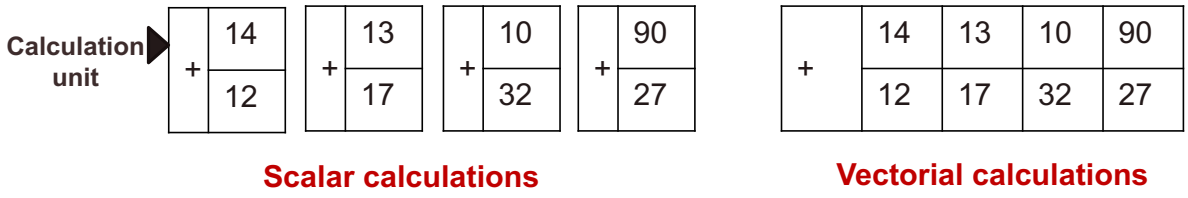

Figure 5. 1: scalar and vectorial calculation units

- **Duplicating the cores or the processors:** Multi-cores in the same die, either through the use of independent core with a common Front Side Bus (FSB) and chipset (data flow management system), or cores with common cache memory (L2/L3). Multi-processors like in supercomputers, achieves the pure parallelism (MIMD). Both (multi-core and multi-processor) allow the parallel executions of threads and apps.

V.3. Multicore and Parallelism

Multi-core Processors: A multi-core processor integrates two or more independent cores into a single computing component. Each core can execute instructions independently, allowing for parallel processing of multiple tasks.

Parallel Processing: Parallel processing involves dividing a computational task into smaller sub-tasks that can be executed simultaneously across multiple cores or processors. This approach can significantly reduce the time required to complete complex computations.

V.4. Advantages and challenges of multi-core architecture

V.4.1. Advantages:

- **Increased Performance**: By distributing workloads across multiple cores, multi-core processors can achieve higher throughput and faster execution times for parallelizable tasks.
- **Improved Energy Efficiency**: Multi-core processors can achieve better performance per watt compared to single-core processors, as they can operate at lower frequencies and voltages.
- **Enhanced Responsiveness**: Multi-core systems can handle multiple tasks simultaneously, improving the responsiveness of applications and the overall user experience.

V.4.2. Challenges:

- **Software Complexity**: Developing software that effectively utilizes multiple cores can be challenging. Parallel programming requires careful consideration of task decomposition, synchronization, and load balancing.
- **Scalability**: As the number of cores increases, the overhead associated with communication and coordination between cores can become a bottleneck, limiting the scalability of multi-core systems.
- **Thermal Management**: Higher core counts can lead to increased heat generation, requiring advanced cooling solutions to maintain optimal operating temperatures.

V.4.3. Measuring CPU performance

- **Response time (latency):** How long does a thing take from when we start something until it's done
- **Throughput:** How many things can we do per second.
- **Iron** Law: $\frac{1}{performance} = \frac{instructions}{program} \times \frac{cycles}{instruction} \times \frac{time}{cycle} = \frac{time}{program}$

  performance $= \frac{program}{time} = \frac{IPC \times frequency}{Instruction\ Count}$

  IPC $= \frac{1}{CPI} = \frac{instructions}{cycle}$

V.4.4. Parallel Processor Performance

- **Amdahl** Law:
- **Efficiency** of parallel machine **E**=1-h× $(1 - \frac{1}{N})$. This metric measure how well the processor is using its resources.

Where h: fraction of time spent on scalar computation; **1-h**: fraction of time spent on vector computation; **N**: number of Processors

- **Speed up, S** measures how fast is the parallel processor in executing a program compared to a sequential processor.

$$\mathbf{S}=\frac{\mathbf{1}}{\boldsymbol{T}}=\frac{\mathbf{1}}{(\mathbf{1}-\boldsymbol{f})+(\frac{\boldsymbol{f}}{\boldsymbol{N}})}$$

Where1-f: time to execute a sequential fraction of program; **f/n:** time to execute a parallel fraction of program

Noting that **efficiency** $=\frac{\boldsymbol{S}}{\boldsymbol{N}}$

- **Comparing two machines' performance**

Machine A is **n** times faster than machine B iff **perf(A)/perf(B)** = **time(B)/time(A)** = **n**

Machine A is **x%** faster than machine B iff **perf(A)/perf(B)** = **time(B)/time(A)** = **1** + **x/100**

V.5. Cache Memory and Memory Hierarchy

The memory hierarchy is a structured organization of different types of memory, each with varying speeds, sizes, and costs. The hierarchy typically includes registers, cache memory, main memory (RAM), and secondary storage (e.g., SSDs, HDDs).

V.5.1. Random Memory Access

Random access memory (RAM) can be used to write or read information. It constitutes the largest part of the main memory of a computer. **RAM** memories are realized with two different technologies:

V.5.1.A. Dynamic RAM (DRAM)

This type of memory is widely used because it is not expensive. The dynamic memory barrettes enclose a silicon chip on which are integrated a very large number of binary cells. Each binary cell is made from a transistor connected to a small capacitor. The charged or discharged state of the capacitor makes it possible to distinguish between two states (bit 0 or bit 1). The disadvantage of this simple technique is that the capacitor discharges itself over time (current leakage). It is therefore necessary to refresh all the capacitors periodically, approximately 1000 times per second. This operation is carried out by an

enclosed refresh circuit integrated. The circuit reads the state of each cell and rewrites it, which recharges the capacitor. Note that this operation prevents access to a memory cell for a few clock cycles.

V.5.1.B. Static RAM (SRAM)

Static memories do not use capacitors: each binary cell is made using 4 transistors forming a bistable, circuit remaining in state 0 or 1 as long as it is electrically powered. SRAMs allow shorter access times than DRAMs but are more expensive because their construction require 4 times more transistors than DRAMs. SRAM are used when you want to maximize performance, for example to build cache memories. Noting also the existence of CMOS SRAM, characterized by their very low energy consumption: a small electric battery is enough to keep them in operation for several years. These memories, of small capacity, are used for example in certain electronic diaries for the general public. SIMM modules (Single In-line Memory Module) are groups of dynamic memory boxes mounted on an elongated rectangular printed circuit, called a bar (see Figure 5.2). Each SIMM bar offers a large capacity, and plugs into connectors provided for this purpose on the computer's motherboard.

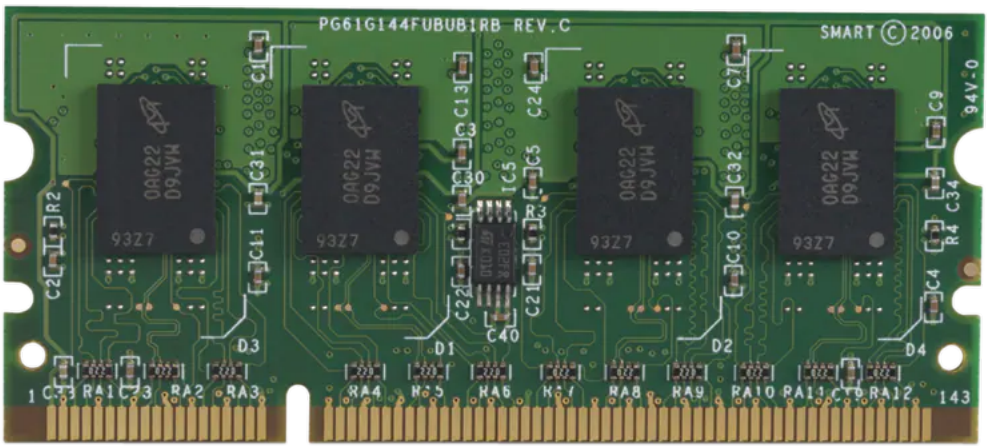

Figure 5. 2 : Memory bar[17]

[17] https://images.app.goo.gl/bNGrSDxuUjodMSbSA

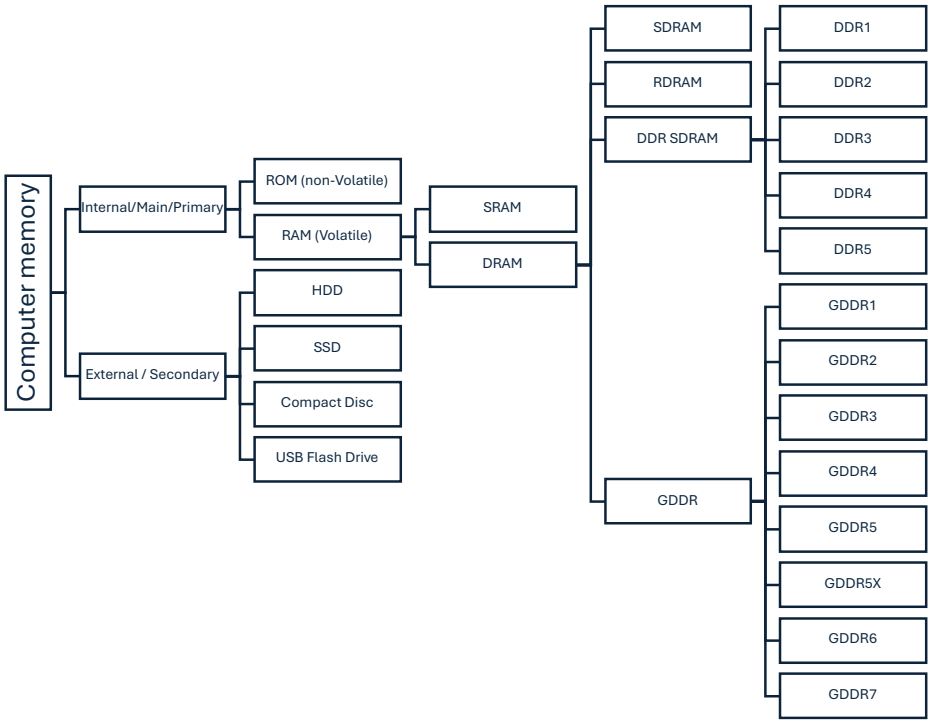

Figure 5. 3: Computer memory types

V.5.2. Read Only Memory

ROMs are normally read-only. There are different types of ROM circuits:

- **ROM**: integrated circuit whose content is determined once and for all at the time of manufacture. Their relatively high cost of manufacturing requires mass production, which complicates the updating of their content. Initially, these memories were used to store the low-level parts of the computer operating system (PC BIOS for example)
- **PROM (Programmable ROM):** While ROM is hard-coded during manufacture, **PROM is user-configured using a PROM programmer,** which is used to save its contents. **The PROM circuit cannot be modified afterwards**. It was used in early computers to store the BIOS (replaced currently by EEPROM)
- **EPROM (Erasable PROM):** EPROM memories are reconfigurable PROMs; it is possible to erase them to reprogram them. Erasure occurs by exposing the case to strong ultraviolet (UV) radiation (refer to Figure 5.4). For this, the case is pierced with a transparent window allowing the exposure of the integrated circuit. The erasing operation requires removing the EPROM from its support and results in immobilization for approximately 30 minutes. Previously used to store the BIOS (replaced currently by EEPROM)

Figure 5. 4: EPROM and its UV eraser[18]

- **EEPROM (Electrically Erasable PROM):** Same principle as an EPROM, but the erasing is done using electrical signals, which is faster and more convenient. An example of where it is used: security gadgets, such as: credit card, SIM card, key-less entry, etc.
- **FLASH EPROM:** FLASH memories are similar to EEPROM memories, but erasing can be done selectively in blocks and does not require dismantling the circuit. The writing time of a block of FLASH memory is much greater than that of writing RAM memory, but of the same order as that of a hard disk. Read access to an EEPROM is about as fast as a DRAM. Flash memory cards are therefore sometimes used as secondary memory

V.5.3. Cache Memory

It is faster than main memory but smaller in size and it keeps information that is supposed to be in main memory that the processor uses most often at a given time. Refer to figure 5.5 for illustration.

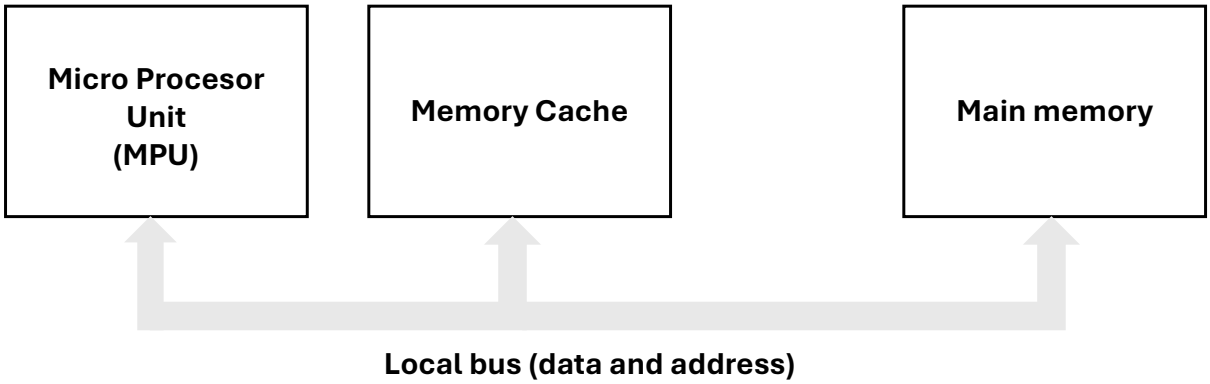

Figure 5. 5: Memory cache integration within a processor

[18] https://images.app.goo.gl/YxQcWMh32yxXeV6q6

When the processor accesses a word in memory, two cases can occur:

1. **the word is present in the cache:** the cache, which is faster, sends the requested data on the data bus.

2. **the word is not present in the cache:** the memory access takes place normally (the cache can take advantage of this to copy the data in order to have it for the next time).

V.5.3.A. Effectiveness of a cache: locality principle

The use of a cache memory is **effective** only if it **frequently** happens that a word requested by the processor is already in the cache. A data **enters** the cache after it was **read** into main memory. The first access to a data item is therefore **slow**, but **subsequent** (following) accesses to the same address will be **faster**. It is easy to see that the more the program you are running makes varied memory accesses, the **less efficient** the cache will be.

The **locality** of a program is its **tendency to perform repeated** memory accesses; this tendency increases the speed gain provided by the cache. There are **two types** of localities the temporal and the spatial. **Temporal** locality indicates that items accessed **recently** will likely be used in the near future. **Spatial** locality indicates that **nearby items** tend to be referenced at nearby times.

V.5.3.B. Cache Organization

The cache is organized by rows (see Figure 5.6 for illustration). Each line contains an 8-to-512-byte portion of data in memory and a tag which is the address of that data in memory. When the microprocessor wants to access the memory, it compares the address with the cache line labels. If it finds the address among the tags, the microprocessor directly uses the data from the cache. This is referred to as a cache hit. Otherwise, it is called cache miss.

Tag	Index	Dirty	Data
36F2	0	0	4F....
05A2	1	0	2A...
0083	2	1	00...
0A2A	3	0	E1....
A3C4	4	1	00...
FFC4	5	1	56..
50DE	6	0	09
0010	7	0	CB..

Figure 5. 6: Direct mapping cache organization

For a better compromise between the price of the cache memory and its efficiency, several levels of cache are used. This is referred to as a cache hierarchy. Most high-performance computers use 3-4 levels of cache (refer to Figure 5.7 for illustration).

- **Level 1** cache is closest to the microprocessor. It is small but made of very fast memory.
- **Level 2** cache is larger in size but made of slower memory. It is often in the same box as the microprocessor which allows a large bus between the two.
- **Level 3** cache is even bigger but still less fast memory.

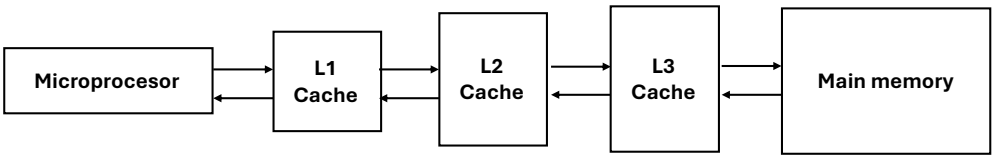

Figure 5. 7: Cache hierarchy

Level 1 cache is often organized into two separate caches, one for **instructions** and another for **data**. Two buses then allow the microprocessor to simultaneously access the two caches. In the case of an architecture with a **pipeline**, this organization is essential to avoid certain **structural hazards**. Also, accesses to instructions or data behave quite differently. Separating the caches allows different optimizations for the two caches (See Figure 5.8).

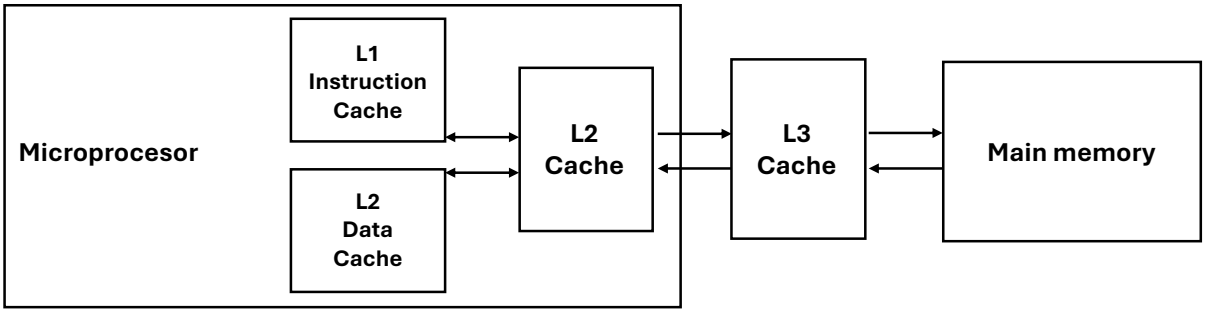

Figure 5. 8: Cache levels with separate L1 caches for data and instruction

V.5.3.C. Cache Miss

The response to a cache miss depends on the nature of the data access. In the case of a read, the data is loaded from the main memory into the cache and then sent to the microprocessor. This loading requires first freeing a line from the cache to place the new data there. Which row to release is controlled by the override policy. Loading data into the cache from main memory involves a delay since the latter is much slower than the cache. This delay is sometimes increased by the fact that it is necessary to copy the content of the freed line into memory. In the case of a **cache miss** during a write, two techniques are used. The **first technique** is to do as for the reading **by loading the data into the cache** and then letting the **processor write into the cache**. The **second technique** is to write the data directly to **memory**. The idea is that during a write, the

processor does not need to wait for it to be completed to continue working. In this case, the writes are put **on hold in a buffer** in order to allow these writes to be carried **out without slowing down the processor** even in the event of several consecutive writes.

When there are too many cache misses, performance plummets (drops down). The cache then operates at the speed of main memory or even slower due to processing overhead. A lot of research has gone into reducing the number of cache misses.

Cache misses are generally classified into **three categories**:

- **Compulsory misses**: when a block is referenced for the first time, it must be loaded into the cache. This type of misses is somehow **unavoidable**.
- **Capacity misses:** These misses are due to the fact that the cache cannot contain all the referenced blocks during the execution of the program. The number of these misses can be reduced by increasing the size of the cache.
- **Conflict misses:** A block could be loaded and then removed from the cache because other blocks with the same index were loaded. The number of these defects can be reduced by increasing the associativity of the cache.

V.5.3.D. Write policies

There are several ways called write policies to manage writes to caches.

- The policy called **write-through** consists of **reflecting** in main memory each write in the cache. i.e. Each write in the cache then causes a write in main memory.
- In contrast, the **write-back** policy **delays writings** to the main memory as much as possible. Data that has been written to cache is written to main memory when the row that contains **that data is freed**. To know if this write is necessary, each line contains a bit called **dirty bit** which indicates if there was at least one write in this line.

There are **also intermediate policies**.

- In the case of the **write-through policy**, writes to be made can be temporarily held in a queue. Several consecutive writes to the same address can thus be passed on by a single write in memory. This mechanism reduces the exchanges with the central memory.
- In the case of a **write-back policy**, the cache can anticipate the writing in memory of certain modified lines of the cache. It takes advantage of periods without swapping with the memory to write certain lines whose dirty bit is set. This technique makes it possible to avoid writing the line when it is freed.

V.5.3.E. Associativity

A crucial element of **cache efficiency** is quickly finding whether data **at a memory address is already in cache**. In order to speed up this search, the number of lines where data can be placed at a fixed memory address is often reduced. This number is not address dependent and is called the **associativity** of the cache. When this number is reduced to 1, i.e. data from each address can be put into a single line of cache, it is called **direct map cache** (refer to Figure 5.9**).** If, on the contrary, the associativity is equal to the number of lines in the cache, i.e. each piece of data can be put in any line of the cache, the cache is said to be **fully associative**. If the associativity is an integer **n**, it is called an **n-associative (n-way) cache**. The caches are very often direct, 2-, 3- or 4-associative but rarely more.

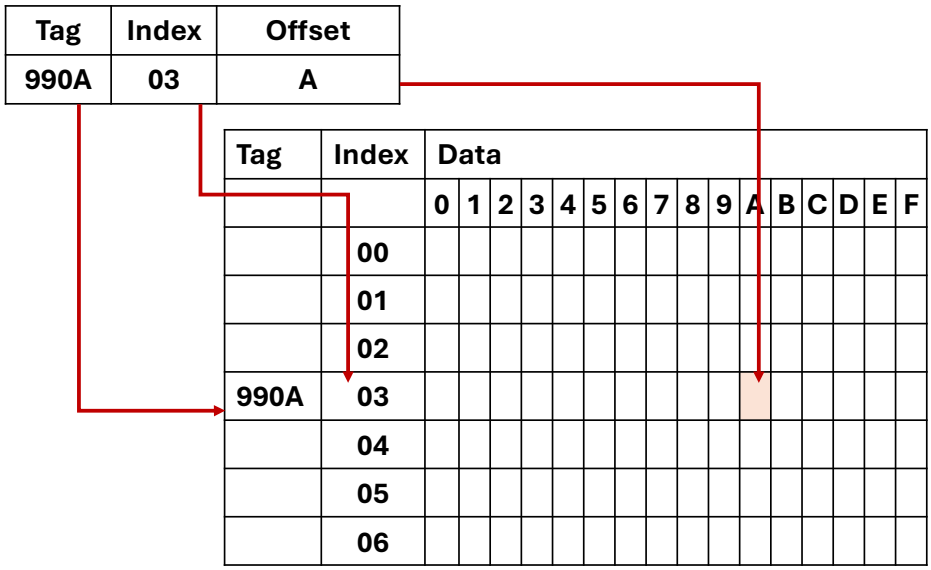

Figure 5. 9: Direct mapping cache

In the case of **n-associative caches**, the cache organization is similar to direct caches. Each address corresponds to **n consecutive cache lines** instead of one. The index of the first line is again determined by low order bits of the address. Memory blocks with the same index are generally called a **set**.

V.5.3.F. Replacement policies

When a cache **miss** occurs, the data must be placed in one of the cache lines determined by the address. It then remains to choose **which line to free** for the new data. There are several ways to make this choice which are called **override policies**. The objective of these policies is to **minimize** the number of cache misses, by trying to better predict the data that will be used by the microprocessor. These different replacement policies have of course a compromise between their performance and the additional computational cost they imply. Since these calculations must be performed at the level of the circuits of

the cache, they are necessarily simple. **These policies favor freeing a line where there has been no writing. This choice saves the writing in central memory of the data contained in the freed line.** The two most commonly used policies are **random draw** and the so-called **Least Recently Used** policy.

- The **random draw** consists of choosing one of the possible lines at random. The advantage of this policy is to minimize the computation overhead while giving reasonable performance.
- The second policy chooses the line to which the **last access is the oldest**. The underlying idea is that this line has a lower probability of being used in the future. In particular, the data which is **no longer used by the microprocessor disappears from the cache**. In fact, it is most often approximations of this policy that are used in practice most commonly used policies are random draw and the so-called Least Recently Used policy.

V.6. Virtual Memory

When compiling a program written in a high-level language, all references to local or global variables are transformed into references to memory locations allocated to these variables. Global variables are usually allocated in the data area while local variables are allocated on the stack instead. **Virtual memory** is a technique for running programs that **are larger than real memory.** In a computer without virtual memory, the addresses manipulated implicitly or explicitly by the programs **correspond to the real addresses of the data in memory**.

The principle of **virtual memory** is to **separate** the **addresses manipulated by the programs** and the **real addresses of the data in memory**. Addresses manipulated by programs are called **virtual addresses** and addresses of data in memory are called **physical addresses**.

Each program has a **virtual address space** consisting of all virtual addresses. For programs to work correctly, there must be a **correspondence** between **virtual addresses** and **physical addresses**. This correspondence is ensured jointly on the one hand at the **hardware** level by an assistant circuit to the processor called **MMU (Memory Management Unit)** and on the other hand at the **software** level by the **operating system**. In recent microprocessors, this MMU circuit is often integrated into the processor.

The **virtual address space** of each program as well as the memory space is divided into blocks called pages of **equal size**. This size is very often around 4 KB but can also be 8 or 16 KB on 64-bit microprocessors. A block of virtual address space is called **a virtual page,** and a block of memory space is called a **physical page**.

For each virtual page used by a program corresponds a physical page. The system maintains for each program a table of **correspondence** between the virtual pages and the physical pages. This table is called the **page table** or mapping table (see Figure 5.10). It is of course **specific** to each program. Two programs running simultaneously can use the **same virtual address**, for example **0x1234**, for **different data** which must therefore be at **different physical addresses**.

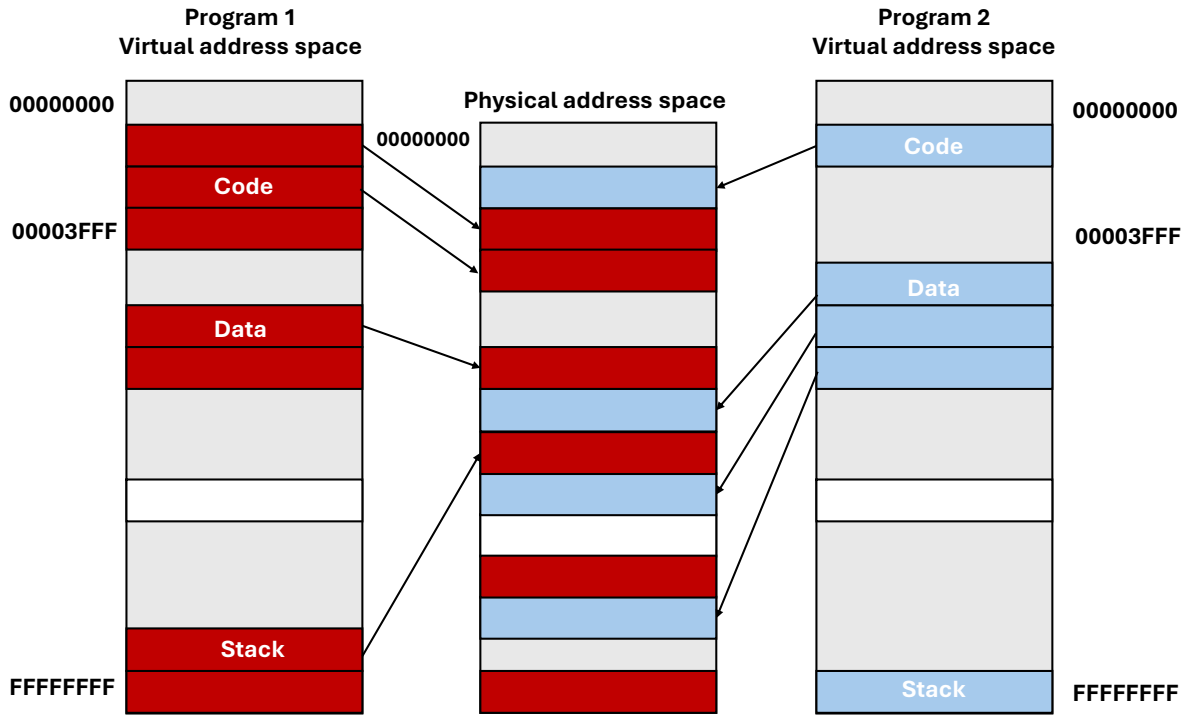

Figure 5. 10: Virtual Memory page table

V.7. Conclusion

In this chapter, we resumed the efforts made to design modern processors to ensure high performance. Starting of the use of vectorial units, duplicating units with the processor, adopting multi-core approach to ensure parallel threads and processes execution, to the multiprocessor approach that ensures real parallelism.

The evolution of computers is not limited to the amelioration of processors, and it expanded to the inclusion of different types of memory relying on various types of technologies such as the SRAM and DRAM. The SRAM is used in cache memory and DRAM is used in main memory. The main memory is supported by virtual memory stored on hard disk (secondary memory).

In the next chapter, we introduce recent computing technologies such as the coprocessors, the embedded systems, and the quantum computers.

V.8. QCM

1. Which of the following best describes modern processor architecture?

a) Single-core, sequential execution only

b) Multi-core with advanced memory management

c) Only uses RISC architecture

d) Limited to 32-bit instruction sets

2. What is the primary purpose of modern processor design improvements?

a) Reduce physical size only

b) Minimize cost of production

c) Increase CPU performance and efficiency

d) Simplify programming languages

3. Which technique allows a CPU to execute multiple instructions in one clock cycle?

a) Pipelining

b) Branch prediction

c) Superscalar execution

d) Time-sharing

4. What does "instruction-level parallelism" refer to?

a) Running multiple programs at once

b) Executing multiple independent instructions simultaneously

c) Increasing cache size

d) Reducing clock speed

5. Which of the following improves performance by predicting the direction of branches in code?

a) Cache prefetching

b) Out-of-order execution

c) Branch prediction

d) Memory paging

6. What is a key advantage of multi-core processors over single-core processors?

a) Lower power consumption

b) Ability to run legacy software faster

c) Improved multitasking and parallel processing

d) Simpler hardware design

7. Which of the following is a major challenge in multi-core systems?

a) Reduced memory capacity

b) Difficulty in software optimization for parallelism

c) Slower clock speeds

d) Incompatibility with 64-bit OS

8. What is Amdahl's Law primarily used to evaluate?

a) Maximum CPU temperature

b) Potential speedup from parallelization

c) Cost vs. performance trade-off

d) Cache hit rate

9. Which level of cache is typically the fastest but smallest in size?

a) L1

b) L2

c) L3

d) RAM

10. What is the main reason for using a memory hierarchy in modern processors?

a) To reduce manufacturing cost

b) To balance speed, cost, and capacity

c) To increase instruction length

d) To simplify CPU design

11. What is a cache hit?

a) When data is not found in cache

b) When data is found in cache

c) When cache needs to be flushed

d) When cache memory is full

12. Which cache mapping technique offers the best flexibility?

a) Direct-mapped

b) Fully associative

c) Set-associative

d) Random mapping

13. What is virtual memory primarily used for?

a) Speeding up cache access

b) Extending available RAM using disk space

c) Improving CPU clock speed

d) Storing BIOS settings

14. Which component translates virtual addresses to physical addresses?

a) ALU

b) MMU (Memory Management Unit)

c) Cache controller

d) DMA controller

15. What is a page fault?

a) When a program tries to access a non-existent page

b) When the cache is full

c) When a process exceeds its time slice

d) When the CPU overheats

16. Which unit of data is swapped between RAM and disk in virtual memory systems?

a) Byte

b) Word

c) Page

d) Sector

17. Hyper-Threading Technology (HTT) helps improve CPU performance by:

a) Cooling the CPU more efficiently

b) Allowing multiple threads to run on a single core

c) Increasing clock speed dramatically

d) Replacing cache memory

18. What is the role of the Translation Lookaside Buffer (TLB)?

a) Store recently used instructions

b) Store frequently accessed data

c) Cache virtual-to-physical address translations

d) Manage I/O operations

19. Which of the following is NOT part of the memory hierarchy?

a) Registers

b) Main memory (RAM)

c) Hard Disk Drive (HDD)

d) Arithmetic Logic Unit (ALU)

20. Which of the following statements about multi-core processors is FALSE?

a) They can execute multiple threads simultaneously

b) They always provide linear performance improvement with added cores

c) They require operating system support

d) They help reduce heat per core compared to a single large core

V.9. QCM Solutions

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
B	C	C	B	C	C	B	B	A	B	B	C	B	B	A	C	B	C	D	B

Chapter 6: Introduction to advanced concepts

VI.1. Introduction

In this chapter, we introduce the difference between general computing and embedded systems that are developed for specific purposes. Then, we take Arduinos as example of embedded systems. Subsequently, we introduce the concept of co-processors such as the graphical card, the cryptography processors and the neural processing units used for artificial intelligence. Lastly, we briefly explain the quantum computers and their functioning.

VI.2. Embedded Systems

In Embedded systems (see Figure 6.1), resources can be implemented on a single IC. ICs include a variety of peripherals such as: timers, analog-to-digital converters, digital-to-analog converters, serial interfaces, etc. Moreover, their small size makes them very versatile. As for the software layer, the IC contains firmware (only the needed software which is not intended to be changed frequently).

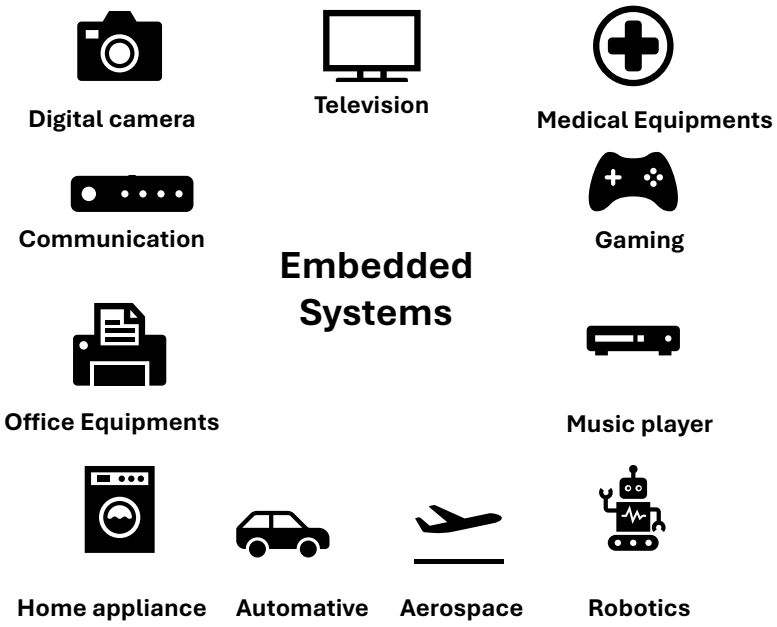

Figure 6. 1: Embedded Systems

An embedded system is a combination of hardware and software that has a dedicated, specific purpose inside devices. They function as application-specific computing systems. Some examples include digital cameras, MP3 players, and kitchen appliances. Sometimes, an embedded systems program works on its own with a set of functions and limited user controls. This can be seen in some appliances, cars, or certain types of medical equipment where users simply need to turn on the device. Other embedded systems are designed with a user interface (UI) or graphical user interface (GUI), such as on phones,

digital watches, or airplanes. A UI or GUI allows the user to control or program the device as needed. The user interface may include simple physical buttons or light-emitting diodes (LEDs).

"In 2009, it was estimated that ninety-eight percent of all microprocessors manufactured were used in embedded systems."

VI.2.1. Characteristics of embedded systems (ES)

- **Task-specific functionality:** Embedded systems perform a specific task, in contrast with general-purpose computers designed for multiple tasks
- **Realtime constraints:** Some ES have real-time performance constraints that must be met, for reasons such as safety and usability
- **Resource constraint:** Other ES may have low or no performance requirements, allowing the system hardware to be simplified.
- **Cost-effective:** ES are often designed to be cost-effective; their design is simplified both the system design and manufacturing processes.
- **Interaction with the physical world:** ES interact with the external environment through sensors (to gather data) and actuators (to perform actions)
- **Reliability and stability:** ES are designed for low maintenance, long-term operation and to be High Reliability.

VI.2.2. Embedded Systems components

The ES are generally composed of a microcontroller (See figure 6.2). The microcontroller contains a processor, a memory and I/O Ports (Interfaces). The microcontroller's processor has an instruction set, memory and accelerators. As for the memory, it has both the volatile memory (DRAM or SRAM) as well as non-volatile memory (ROM, EPROM, EEPROM, Flash). For the interfaces, ES may have hardware interfaces (ports) such as parallel, serial, and analog. The software interfaces represent the drivers.

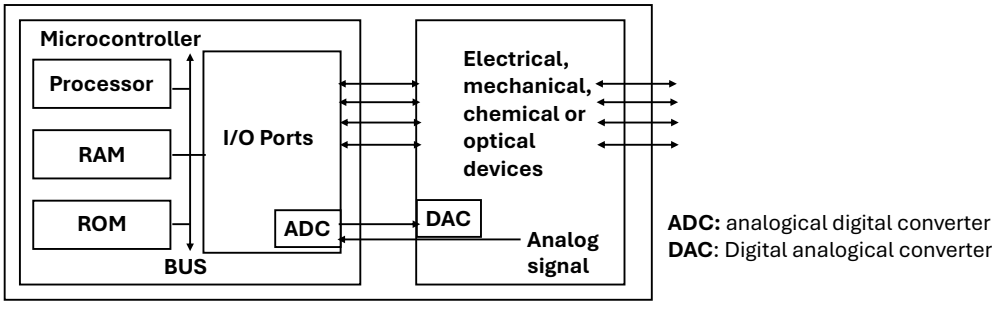

Figure 6. 2: Embedded Systems components

VI.2.3. Product Life Cycle

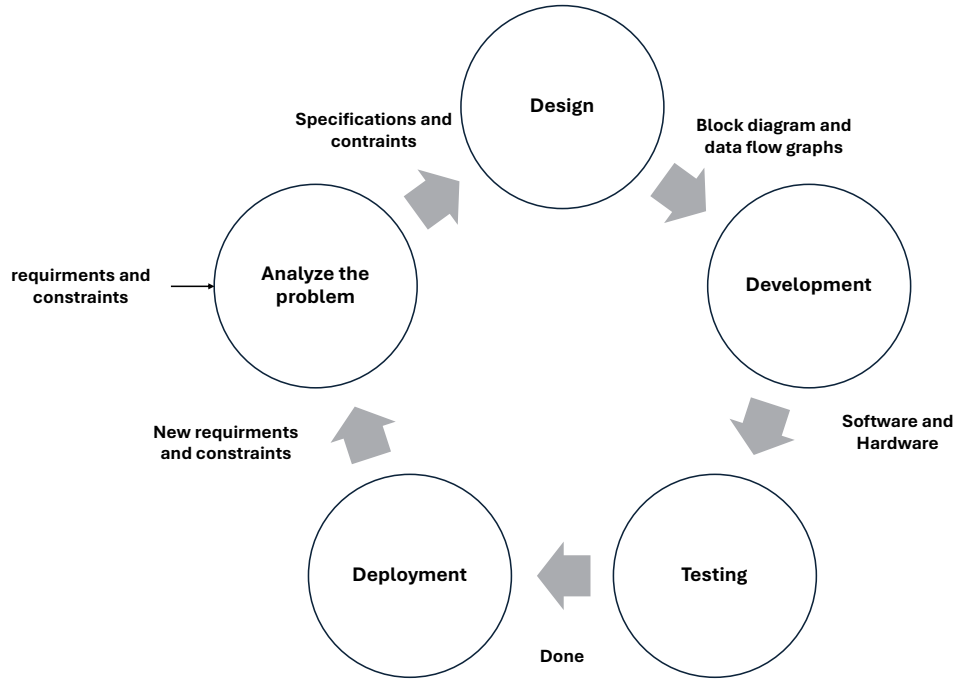

Figure 6. 3: Embedded Systems life cycle

- **Analysis (What?):** Define system requirements and translate them into precise specifications based on functionality, constraints, and performance needs.
- **Design (How?)**
- **High-Level Design:** Create block diagrams, define system architecture, and identify major hardware/software components.
- **Engineering Design**: Develop algorithms, choose appropriate data structures, and define interfaces between hardware and software modules.
- **Implementation (Realization):** Develop and integrate both hardware (e.g., microcontrollers, sensors, communication modules) and software (firmware, drivers, application logic).
- **Testing (Validation)**
- **Validation Testing:** Ensure functional correctness through unit, integration, and system testing.
- **Performance Testing:** Evaluate system efficiency, latency, power consumption, and reliability.
- **Maintenance (Enhance & Evolve):** Apply updates, fix issues, and improve the system post-deployment to adapt to new requirements or optimize performance.

VI.2.4. Arduino

Arduino is an open-source electronics platform based on simple software and hardware. It's used for building a variety of digital devices and interactive objects that can sense and control the physical world. Arduino is a popular platform for **embedded systems** development, particularly for beginners and hobbyists due to its simplicity, accessibility, and large community support. Arduino, while not as "industrial-grade" as some other embedded system platforms (e.g., ARM-based microcontrollers), is an excellent choice for **prototyping and learning** embedded system concepts

The reasons why Arduino is widely used for embedded systems:

- **Ease of Use**: The Arduino platform abstracts much of the complexity of embedded systems development. The programming environment and language are simple and easy to get started with.
- **Real-Time Operation**: Although Arduino doesn't have a full-fledged real-time operating system (RTOS) by default, it can still handle simple real-time tasks through careful programming (e.g., using timers and interrupts).
- **Hardware Flexibility**: Arduino boards come with a variety of sensors, actuators, and communication interfaces, making them highly adaptable for embedded applications.
- **Low Power**: Arduino boards like the **Arduino Pro Mini** or **Arduino Nano** can run on low power, making them suitable for battery-operated devices.
- **Extensive Library Support**: Arduino has a wide range of libraries to interface with different hardware components (sensors, motors, displays, etc.), reducing development time and complexity.
- **Community Support**: A large community of developers, engineers, and hobbyists make it easy to find resources, tutorials, and solutions to common issues.

VI.2.4.A. Key Components of Arduino:

- **Arduino Board (Hardware)**: The board contains a microcontroller that you can program to interact with sensors, motors, LEDs, and other components. Popular boards include:
    1. **Arduino Uno**: A popular starter board based on the ATmega328P microcontroller.
    2. **Arduino Nano**: A smaller, more compact board.
    3. **Arduino Mega**: A larger board with more input/output (I/O) pins for complex projects.
- **Sensors and Actuators**: Arduino can interface with a variety of sensors (e.g., temperature, light, motion) and actuators (e.g., motors, LEDs) to build interactive systems

- **IDE (Integrated Development Environment)**: The Arduino IDE is where you write, edit, and upload your code (called "sketches") to the Arduino board. It's a free software that supports both Windows, Mac, and Linux platforms.
- **Programming Language**: Arduino uses a simplified version of C/C++ to write the code. It also includes several libraries that make it easier to interact with various components (e.g., motors, sensors, displays).

VI.2.4.B. Basic Concepts:

- **Inputs and Outputs**: Arduino boards have pins that can be set to input or output mode. Input pins receive data from external sensors, while output pins send signals to control devices like LEDs or motors.
- **Digital and Analog**: Some pins are digital (ON/OFF, 0 or 1), and others are analog (continuous range, like voltage levels).
- **Pulse Width Modulation (PWM)**: This technique allows you to simulate analog output using a digital pin by rapidly switching the output between HIGH and LOW.

VI.3. General-Purpose Computers

The general-purpose computers- GPC (see Figure 6.4) are able to run a variety of software. They contain relatively high-performance hardware components (fast processors, data & program storage). Also, they require an operating system (OS). GPC are designed for heavy user interaction. They use a variety of peripherals (displays, keyboards, mice, internet connections, wireless communication capability). Thus, they are expensive. In term of design, they use a group of integrated circuits or chips (ICs). One implements the central processing unit (CPU) while several implement data memory and program storage. Moreover, they possess a distributed architecture (full functionality of the computer is spread across multiple IC chips). Figure 6.5 depicts the embedded systems, the general purpose computers and their intersection.

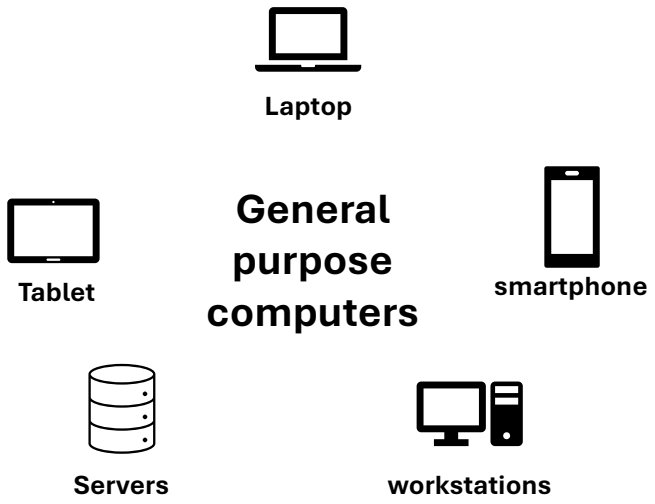

Figure 6. 4: general purpose computers

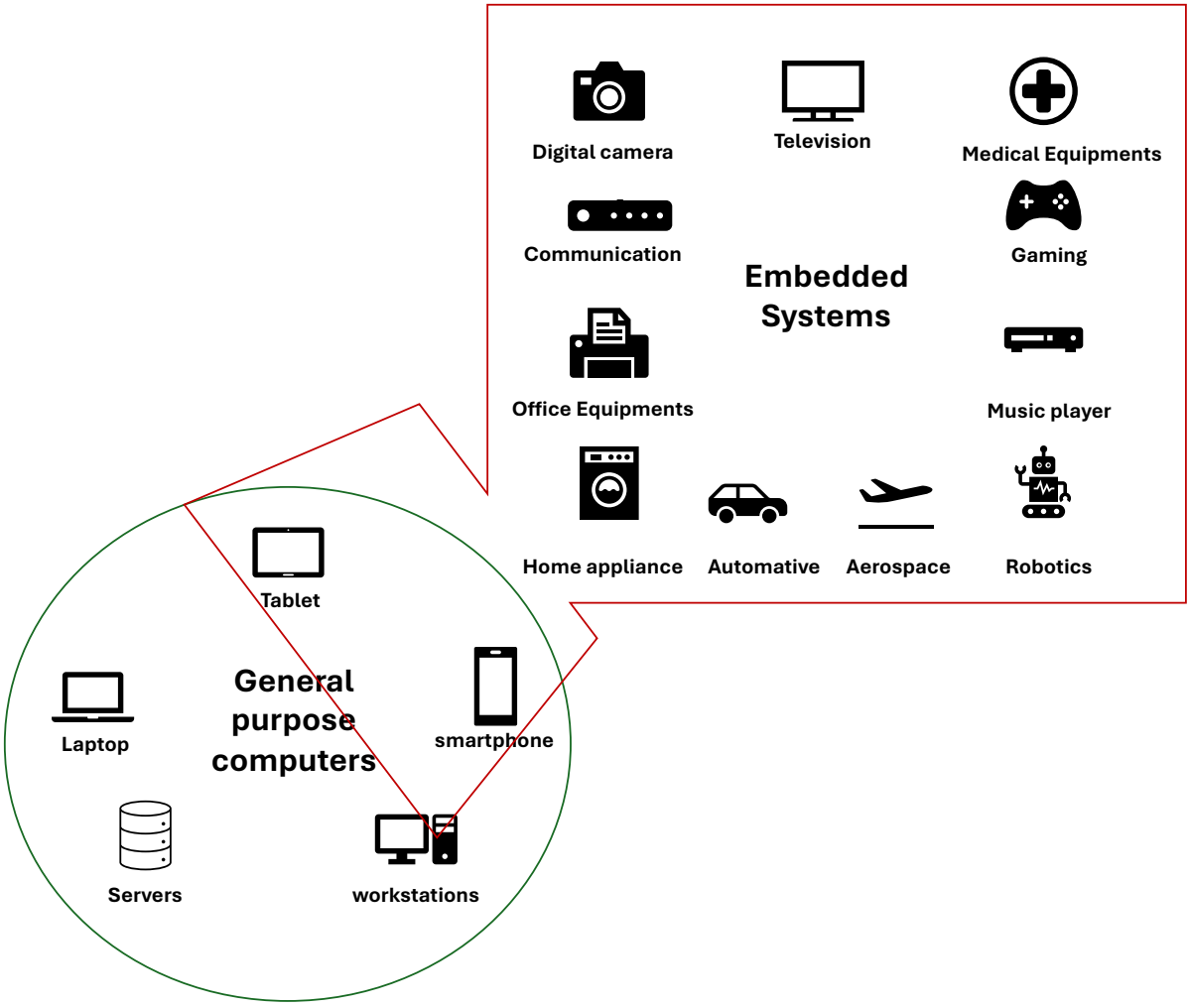

Figure 6. 5: general purpose computers intersection with embedded systems

VI.4. Integrated systems

An **integrated system** typically refers to a combination of subsystems that are designed to work together as a unified whole. The subsystems might be software, hardware, or a mix of both, and the term is often used in the context of systems that are built by integrating various components into a cohesive system. Examples: An integrated smart home system (lighting, heating, security), or an integrated circuit (IC) that combines multiple electronic components (transistors, resistors) into a single chip.

VI.1. Co-Processors

Coprocessors are specialized processors designed to assist the main processor (CPU) by offloading specific tasks, improving overall system performance. They can handle operations that are computationally expensive or specialized, freeing up the CPU for other tasks.

VI.2. AI Accelerators and NPU

An **NPU** is a specialized coprocessor designed for accelerating AI/ML workloads, particularly deep learning and neural network inference. It performs matrix multiplications and other tasks critical for AI algorithms.

Example: Google Tensor Processing Unit (TPU),

Tensor Processing Unit (TPU): Google Cloud TPUs are custom-designed AI accelerators, which are optimized for training and inference of large AI models.

Cloud TPUs VS GPUs: A GPU is a specialized processor originally designed for manipulating computer graphics. Their parallel structure makes them ideal for algorithms that process large blocks of data commonly found in AI workloads. A TPU is an application-specific integrated circuit (ASIC) designed by Google for neural networks. TPUs possess specialized features, such as the matrix multiply unit (MXU) and proprietary interconnect topology that make them ideal for accelerating AI training and inference.

In mathematics, a **tensor** is a multi-dimensional generalization of scalars, vectors, and matrices. More formally, it is a geometric object that describes linear relationships between sets of algebraic objects (such as vectors and scalars) related to a vector space.

- **Scalar**: A tensor of **rank 0** (a single number).
- **Vector**: A tensor of **rank 1** (a list or array of numbers with a specific direction and magnitude).
- **Matrix**: A tensor of **rank 2** (a two-dimensional array).
- **Higher-Rank Tensors**: For tensors of rank 3 or higher, you have multi-dimensional arrays (like 3D arrays, 4D arrays, etc.).

In **machine learning** and **deep learning**, the term "tensor" is commonly used to describe **multi-dimensional arrays** of data. Libraries like **TensorFlow** (from Google) and **PyTorch** (from Facebook) use the concept of tensors as their fundamental data structure for representing and manipulating data.

- **1D tensor**: A **vector** (e.g., a list of numbers).
- **2D tensor**: A **matrix** (e.g., a table or grid of numbers).
- **3D tensor**: A **3D array** (e.g., a stack of matrices, such as a color image with width, height, and color channels).
- **4D tensor**: Often used to represent a **batch of 3D data** in neural networks, such as a batch of color images, where dimensions are (batch size, height, width, channels).
- **5D+ tensors**: Can represent even higher-dimensional data, such as videos or higher-order structures in neural networks.

VI.3. Graphics Processing Unit (GPU)

A **GPU** is a highly parallel processor designed to accelerate graphics rendering tasks. It is a specialized processor designed to accelerate rendering images, video, and animations, and to handle complex mathematical computations for various applications, particularly in graphics and parallel processing tasks. While initially designed for rendering graphics in video games, GPUs are now used in a wide range of applications, from deep learning to cryptocurrency mining, due to their high parallel processing capabilities.

VI.3.1. Key Features of a GPU

- **Parallel Processing**: GPUs are designed to process many tasks simultaneously (in parallel), which makes them extremely efficient for operations involving large datasets or complex calculations. This is in contrast to CPUs, which are optimized for sequential processing. A GPU consists of hundreds or thousands of smaller cores that can perform parallel operations, making them well-suited for graphics rendering and modern computational tasks (e.g., machine learning).

- **Graphics Rendering**: The primary function of a GPU is to handle the calculations required to render images and videos. This involves processing data for textures, shading, lighting, and object movement, among other tasks. GPUs utilize dedicated memory (VRAM) to store textures, frame buffers, and other graphics-related data.
- **High Throughput**: GPUs are optimized for high throughput, meaning they can handle large volumes of data quickly. This is crucial for rendering high-quality graphics in real-time or performing intensive computations
- **Programmability:** Modern GPUs can be programmed to perform custom calculations beyond graphics processing. For example, **CUDA (Compute Unified Device Architecture)** for NVIDIA GPUs allows developers to write general-purpose code that runs on the GPU, enabling applications in fields like scientific computing, machine learning, and data analytics.

VI.3.2. History of GPUs

- **Early Days**: The first GPUs were simply accelerators for 2D graphics, but as demand for 3D rendering and complex visual effects grew, GPUs evolved to handle more sophisticated tasks.
- **NVIDIA GeForce 256** (1999): This was one of the first GPUs that could be marketed as a "GPU," with hardware support for 3D rendering and a dedicated graphics pipeline.
- **CUDA and General-Purpose Computing**: In the 2000s, GPUs began being used for general-purpose computing, which allowed them to perform tasks traditionally handled by CPUs, such as scientific simulations and machine learning.

VI.3.3. How a GPU Works

- **Rendering** Pipeline: A GPU uses a **rendering pipeline** to convert 3D data (e.g., vertices, textures, lighting information) into 2D images that can be displayed on a screen. The main stages of the pipeline include:
  - **Vertex Processing**: Transforming 3D coordinates into 2D coordinates.
  - **Rasterization**: Converting 3D data into a 2D image (pixels).
  - **Fragment Processing**: Applying color, texture, and lighting to each pixel.
  - **Pixel Output**: Writing the final image to the display.
- **Shader Programming**: **Shaders** are small programs that run on the GPU, responsible for processing individual vertices or fragments (pixels). Common types of shaders include:
  - **Vertex Shader**: Modifies vertex data (position, color, etc.).
  - **Fragment (Pixel) Shader**: Calculates the color of each pixel.
  - **Compute Shader**: Used for general-purpose computations outside of the traditional graphics pipeline.

- **Memory Hierarchy**:
  - **VRAM (Video RAM)**: A dedicated high-speed memory used to store textures, frame buffers, and other graphics data. VRAM is optimized for large, high-bandwidth access patterns required by GPUs.
  - **Cache**: GPUs often use caches (e.g., L1 and L2 cache) to speed up access to frequently used data.
- **Parallelism**: GPUs consist of many smaller cores designed for parallel processing, as opposed to the few powerful cores found in a CPU. This allows the GPU to handle massive amounts of data simultaneously. For example, while a CPU may have 4—16 cores, a GPU can have thousands of smaller cores.

VI.3.4. Types of GPUs:

- **Integrated GPUs**: These are built into the CPU or motherboard and share memory with the system. Integrated GPUs are less powerful but sufficient for basic tasks like web browsing, video playback, and light gaming. **Examples:** Intel UHD Graphics, AMD Radeon Vega integrated graphics.
- **Discrete GPUs**: These are separate from the CPU and come with their own dedicated VRAM. Discrete GPUs are much more powerful and are designed for demanding tasks like gaming, 3D rendering, and scientific simulations. **Examples:** NVIDIA GeForce, AMD Radeon RX series

VI.3.5. Major GPU Manufacturers:

- **GPUs for Data Centers and High-Performance Computing (HPC)**: Specialized GPUs designed for parallel computing tasks beyond graphics, such as machine learning, artificial intelligence (AI), and scientific simulations. These GPUs are often used in **data centers** and **supercomputing environments**. **Examples**: NVIDIA Tesla, NVIDIA A100, AMD Instinct MI series.
- **NVIDIA**: NVIDIA is the dominant player in the GPU market, particularly for gaming and professional graphics. NVIDIA's GPUs are known for high performance, and their **CUDA** technology is widely used for general-purpose GPU (GPGPU) computing. Popular series: **GeForce** (gaming), **Quadro** (professional), **Tesla/A100** (HPC and AI).
- **AMD**: AMD's GPUs are known for offering a good balance of price and performance. AMD's **Radeon** series is popular for gaming, and their **Radeon Instinct** series is used for data center and HPC tasks. AMD's GPUs also support **OpenCL**, an open standard for parallel programming.
- **Intel**: Intel has been a newcomer to the discrete GPU market, with its **Intel Arc** series targeting gamers and content creators, and future products targeting AI and data center markets.

- **Other Manufacturers**: **ARM** (Mali GPUs) and **Imagination Technologies** (PowerVR) also produce GPUs, mainly for mobile and embedded devices.

VI.4. Cryptographic co-processors

These coprocessors are dedicated to accelerating cryptographic operations such as encryption, decryption, and hashing. They are typically used in applications where security is critical.

Usage: Used for secure communications, digital signatures, blockchain, and payment systems. **Example:** Hardware Security Modules (HSM), specialized cryptographic accelerators.

Interface: Can be PCIe cards, USB devices, or network-attached devices.

VI.4.1. Key Features and Benefits:

- **Performance**: Cryptographic algorithms, especially those used in modern encryption standards (e.g., AES, RSA, ECC), can be computationally intensive. A dedicated co-processor accelerates these operations, significantly reducing the time it takes to perform them.
- **Security**: These devices are often designed to be tamper-resistant, meaning they are harder to attack compared to software-based encryption. Hardware-based encryption is generally more secure because it isolates cryptographic keys and operations from potential software vulnerabilities or malware.
- **Efficiency**: Offloading cryptographic operations to a co-processor reduces the workload on the main CPU, freeing it to perform other tasks. This results in better overall system performance.
- **Compliance and Standards**: Many cryptographic co-processors are built to comply with international standards, such as FIPS 140-2 (Federal Information Processing Standard for cryptographic modules) or Common Criteria. This makes them suitable for applications requiring high levels of security.

VI.4.2. Hardware Security Modules (HSMs):

HSMs are physical devices used to generate, store, and manage cryptographic keys. They are often used in enterprise environments to secure sensitive data, such as digital signatures, authentication tokens, and encryption keys. HSMs are widely used in sectors like banking, telecommunications, and government to protect the most critical cryptographic functions.

VI.4.3. Smart Cards and Tokens:

These are small, portable devices used for secure authentication and encryption. They often include a microprocessor with cryptographic capabilities, such as RSA key pair generation or digital signature verification. Examples: USB security keys (e.g., YubiKey), SIM cards in mobile phones, or credit card-sized smart cards for identity verification.

VI.4.4. TPM (Trusted Platform Module):

TPMs are specialized microchips embedded in many modern computers and servers to provide hardware-based security functions. They are often used for tasks like secure booting, storage encryption (e.g., BitLocker in Windows), and key management. TPMs provide secure key storage and operations such as signing and hashing, typically as part of platform security solutions.

VI.4.5. Cryptographic Accelerators:

These are often integrated into general-purpose processors or as standalone devices designed to accelerate specific cryptographic operations like AES encryption, SHA hashing, or public key cryptography (e.g., RSA, ECC). For example, Intel's **AES-NI** (Advanced Encryption Standard New Instructions) is a set of instructions embedded in many Intel processors that accelerates AES encryption and decryption.

VI.4.6. FPGAs (Field-Programmable Gate Arrays):

FPGAs can be used as programmable cryptographic accelerators, where the cryptographic algorithms are implemented at a low hardware level, offering high performance for custom cryptographic protocols. FPGAs provide flexibility in hardware design and can be reprogrammed to support new or evolving cryptographic standards.

VI.4.7. Use Cases:

- **Data Encryption and Decryption**: Cryptographic co-processors are commonly used in securing communications (e.g., SSL/TLS), disk encryption (e.g., full disk encryption), and secure cloud storage.
- **Digital Signatures**: Cryptographic co-processors help in signing transactions or documents securely by generating and storing the private keys.
- **Key Management**: Cryptographic co-processors securely generate, store, and manage cryptographic keys, which are crucial for securing communications and data.

- **Authentication**: Hardware-based cryptographic devices, like smart cards or USB tokens, can be used for two-factor authentication (2FA) or to store digital certificates for secure logins.

VI.4.8. Examples of Cryptographic Co-processors:

- **Thales HSMs** – These provide secure key management, data encryption, and digital signatures for various industries like finance, government, and telecommunications.
- **Gemalto SafeNet HSM** – A popular hardware security module used for securing applications, payment systems, and digital certificates.
- **Amazon Web Services (AWS) CloudHSM** – A cloud-based HSM service that provides cryptographic key storage and management for cloud applications.

VI.5. Quantum Computers

A **quantum computer** is a type of computer that leverages the principles of quantum mechanics to perform calculations that would be infeasible or take an impractically long time for classical computers to handle. Unlike classical computers, which process information in binary form (0s and 1s), quantum computers use quantum bits or **qubits** that can represent and store information in multiple states simultaneously.

VI.6. Key Concepts in Quantum Computing

- **Qubits (Quantum Bits)**:

The fundamental unit of information in quantum computing is the **qubit** (quantum bit), which is different from the binary bit used in classical computers. A classical bit can exist in one of two states: 0 or 1. In contrast, a qubit can be in a **superposition** of both 0 and 1 simultaneously, allowing quantum computers to process more information at once. A qubit can also be entangled with another qubit, which leads to **quantum entanglement**, enabling faster and more complex computations.

- **Superposition**:

Superposition is a quantum phenomenon where a qubit can exist in a combination of both the 0 and 1 states simultaneously. This allows quantum computers to explore many possible solutions to a problem at the same time. Mathematically, a qubit can be described as a linear combination of the states $|0\rangle$ and $|1\rangle$, with complex coefficients determining the probability of measuring the qubit as 0 or 1 when measured.

- **Entanglement**:

Quantum entanglement is a phenomenon where two or more qubits become correlated such that the state of one qubit can instantaneously affect the state of another, no matter the distance between them. This allows quantum computers to perform certain operations more efficiently, as measurements of entangled qubits can provide information about multiple states simultaneously.

- **Quantum Interference**:

Quantum interference refers to the ability of quantum algorithms to combine the probabilities of different outcomes in a way that enhances the likelihood of correct solutions while diminishing the incorrect ones. This is a key feature of quantum algorithms that provides a computational advantage.

- **Quantum Gates and Quantum Circuits**:

Quantum gates are the basic building blocks of quantum algorithms, analogous to classical logic gates (AND, OR, NOT). These gates manipulate qubits by changing their quantum state. A sequence of quantum gates forms a **quantum circuit**. Quantum gates operate on qubits in superposition, manipulating their state and entangling them to perform complex computations

- **Quantum Speedup**:

One of the most exciting aspects of quantum computing is the potential for **quantum speedup**. This refers to the ability of quantum computers to solve certain problems exponentially faster than classical computers. Problems like factoring large numbers, simulating quantum physics, and optimization problems could be solved much more efficiently with a quantum computer.

VI.7. Conclusion

Computers continue to evolve to achieve the performance needed by its different users. Future applications deal with huge amount of data and require fast processing and computing capacities. This is leading the researchers and manufacturers to think of potential replacements to current semiconductor-based processors. Even though currently these efforts are in experimentation phase, current results are promising in making quantum computing chips and computers in the future.

VI.8. QCM

1. Which of the following best defines an embedded system?

a) A general-purpose computer used for office work

b) A computer system designed for a specific control function within a larger system

c) A high-performance server

d) A cloud-based computing platform

2. Which of the following is NOT typically a characteristic of embedded systems?

a) Real-time operation

b) Low power consumption

c) High processing power and large memory

d) Dedicated functionality

3. Which processor architecture is commonly used in embedded systems due to its low power consumption?

a) x86

b) ARM

c) PowerPC

d) SPARC

4. What does NPU stand for in modern processors?

a) Network Processing Unit

b) Neural Processing Unit

c) Numeric Processing Unit

d) Next-generation Processing Unit

5. Why are AI accelerators used in modern computing devices?

a) To replace CPUs entirely

b) To improve performance for machine learning and AI tasks

c) To reduce screen latency

d) To manage network traffic more efficiently

6. What is the primary purpose of a GPU?

a) Managing file systems

b) Performing complex mathematical computations in parallel, especially for graphics

c) Storing large amounts of data

d) Handling basic arithmetic operations

7. Which type of computing benefits most from GPU acceleration?

a) Sequential processing

b) Text editing

c) Machine learning and image rendering

d) Simple database queries

8. Which architecture allows GPUs to process thousands of threads simultaneously?

a) Superscalar

b) SIMD (Single Instruction, Multiple Data)

c) MIMD

d) VLIW

9. What is the main function of a cryptographic co-processor?

a) Speed up general-purpose computations

b) Improve video decoding performance

c) Handle encryption and decryption operations securely and efficiently

d) Manage input/output operations

10. Which of the following security features is often supported by hardware cryptographic modules?

a) Secure boot

b) Full-disk encryption

c) Random number generation

d) All of the above

11. What is a Trusted Platform Module (TPM)?

a) A software-based encryption tool

b) A cryptographic co-processor that ensures hardware-level security

c) A type of GPU

d) A memory management unit

12. What is the basic unit of information in quantum computing?

a) Byte

b) Bit

c) Qubit

d) Word

13. How do qubits differ from classical bits?

a) Qubits can be only 0 or 1

b) Qubits can exist in superposition of 0 and 1

c) Qubits are slower than classical bits

d) Qubits use analog signals only

14. Which phenomenon allows qubits to be linked so that the state of one instantly influences another?

a) Parallelism

b) Entanglement

c) Encryption

d) Virtualization

15. Which of the following is a potential application of quantum computing?

a) Breaking current encryption algorithms

b) Improving web browsing speed

c) Replacing traditional databases

d) Enhancing word processing

16. Which of the following uses specialized hardware to offload tasks from the CPU?

a) GPU

b) NPU

c) Cryptographic co-processor

d) All of the above

17. Which device is most likely to include both a CPU and a GPU on the same chip?

a) Smartwatch

b) Microcontroller

c) System on Chip (SoC)

d) Hard drive

18. Which of the following is a challenge in quantum computing today?

a) Too much memory

b) Maintaining qubit stability (decoherence)

c) Lack of programming languages

d) Excessive heat from GPUs

19. Which of the following is NOT a typical component of an embedded system?

a) Microcontroller

b) Sensors

c) High-end GPU

d) Actuators

20. Which of the following technologies relies heavily on parallel processing capabilities?

a) Word processors

b) Web browsers

c) Artificial Intelligence and Deep Learning

d) Command-line interfaces

VI.9. QCM Solution

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
B	C	B	B	B	B	C	B	C	D	B	C	B	B	A	D	C	B	C	C

Exercises (Tutorials)

Exercise 0:

1. Find the binary equivalent of **(D3E.A6)16** and the hex equivalent of **(1011001110.011011101)2**.
2. Find the octal equivalent of **(BD.C4)16** and the hex equivalent of **(565.013)8**

Exercise 1

What is a load-store architecture and what are the advantages/disadvantages of such an architecture compared to other GPR (General Purpose Register) architectures?

Exercise 2

What are the design trade-offs between a large register file and a large data cache?

Exercise 3

Review and describe Apple chips from technical and architectural perspectives. What is the difference between SoC and SiP?

Exercise 4

- **First Generation Computers used:**

a) Transistors b) Vacuum tubes c) Integrated circuits d) Microprocessors

- **Which of the following is NOT a function of a computer?**

a) Input b) Processing c) Outputd) Communication with other systems

- **Which of the following best describes the Harvard Architecture?**

a) Single memory for both data and instructions

b) Separated memory for data and instructions, allowing parallel processing

c) A single CPU with no memory

d) Memory that uses magnetic tapes for storage

- **What is the primary function of the Arithmetic Logic Unit (ALU) in a processor?**

a) Fetch instructions from memory

b) Perform mathematical and logical operations

c) Control the flow of data between the CPU and memory

d) Store instructions temporarily for fast access

- **What type of memory is used for the temporary storage of data while the computer is powered on?**

a) ROM b) RAM c) Cache memory d) Secondary memory

- **Which of the following is the main advantage of using microprocessors in computers?**

a) Increased reliability and reduced size b) Increased complexity

c) Greater need for cooling d) Reduced speed of processing

- **What is the main difference between primary and secondary memory?**

a) Secondary memory is temporary, while primary memory is permanent.

b) Primary memory is faster, while secondary memory is slower.

c) Secondary memory is volatile, while primary memory is non-volatile.

d) Primary memory is used for input and output, while secondary memory is for processing.

- **Which of the following components is responsible for directing operations within the CPU?**

a) Control Unit (CU) b) Arithmetic Logic Unit (ALU)

c) Memory d) Registers

- **Which architecture uses separate memory for instructions and data?**

a) Von Neumann Architecture

b) Harvard Architecture

c) RISC Architecture

d) CISC Architecture

- **The superscalar approach can be used on ..... architecture.**

A. RISC B. CISC C. neither RISC nor CISC D. both RISC and CISC

- **The essence of the ..... approach is the ability to execute instructions independently and concurrently in different pipelines.**

A. scalar B. branch C. superscalar D. flow dependency

- **Which of the following is a fundamental limitation to parallelism with which the system must cope?**

A. procedural dependency B. resource conflicts C. anti-dependency D. all of the above

- **The situation where the second instruction needs data produced by the first instruction to execute is referred to as .....**

A. true data dependency B. output dependency C. procedural dependency D. anti-dependency

- **The instructions following a branch have a ..... on the branch and cannot be executed until the branch is executed.**

A. resource dependency B. procedural dependency C. output dependency D. true data dependency

Exercise 5

For the following assembly instructions, identify the addressing mode used:

1. MOV R1, #5
2. ADD R2, [R3]
3. LDR R1, [R2 + 4]
4. MOV R1, [0x2000]

Exercise 6

Consider the following instruction: MOV R1, #50

1. What is the operand here?
2. What addressing mode is being used?
3. What is the value of register R1 after execution?

Exercise 7

Consider the following instruction: ADD R1, R2, R3

1. What are the operands for this instruction?
2. What addressing mode is being used?
3. What is the result of the addition?

Exercise 8

Consider the following instruction: MOV R1, [2000]

1. What is the operand here?
2. What addressing mode is being used?
3. What value is loaded into register R1?

Exercise 9

Consider the following instruction: MOV R1, [R2]

1. What is the operand here?
2. What addressing mode is being used?
3. If the value in R2 is 1000, what is the effective address?

Exercise 10

Consider the following instruction: MOV R1, [R2 + 5]

1. What is the effective address?
2. What addressing mode is being used?

Exercise 11

Consider the following instruction: MOV R1, [R2 + 100]

1. What is the base address?
2. What is the offset?
3. What addressing mode is being used?

Exercise 12

Consider the following instruction: JMP [PC + 200]

1. What is the effective address of the jump target?
2. What addressing mode is being used?

Exercise 13

Consider the following instruction: MOV R1, [R2 + 8]

1. What is the effective address?
2. What addressing mode is being used?

Exercise 14

Consider the following operation: PUSH R1

1. Where is the operand stored?
2. What addressing mode is being used?

Exercise 15

1. Identify the RAW, WAR, and WAW dependencies in the following instruction sequence:

   I1: R1 = 100

   I2: R1 = R2 + R4

   I3: R2 = r4 – 25

   I4: R4 = R1 + R3

   I5: R1 = R1 + 30
2. Rename the registers from part (a) to prevent dependency problems. Identify references to initial register values using the subscript "a" to the register reference.

Exercise 16

Consider following assembly-language program:

1: MOV R3, R7

2: LOAD R8, R3 // **R8<- Memory content of @ stored in R3**

3: ADD R3, R3, 4

4: LOAD R9, R3

5: BNE R8, R9, L3 // **BNE: Branch if Not Equal : if R8 ≠ R9), branch to L3**

a) This program includes WAW, RAW, and WAR dependencies. Show these?

b) What is the difference between a dependency and hazard?

c) What is the difference between a name dependency and a true dependency?

d) Which of WAW, RAW, WAR are true dependencies and which are name dependencies?

Note: **WAW**: Write After Write, **RAW**: Read After Write, **WAR**: Write After Read

Exercise 17

Identify all data dependencies in the following code, assuming that we are using the 5-stage pipelined datapath. Which dependencies can be resolved via forwarding?

ADD R2, R5, R4

ADD R4, R2, R5

SW R5,100 (R2) //**sw: store a value of register into a memory**

ADD R3, R2, R4

Exercise 18

Consider executing the following code on the 5-stage pipelined datapath: Which registers are read during the fifth clock cycle, and which registers are written at the end of the fifth clock cycle? Consider only the registers in the register file i.e., R1, R2, R3, etc.

ADD R1, R2, R3

ADD R4, R5, R6

ADD R7, R8, R9

ADD R10, R11, R12

ADD R13, R14, R15

Exercise 19

Assume an instruction set that uses a fixed 16-bit instruction length. Operand specifiers are 6 bits in length. There are 5 two operand instructions and 33 zero operand instructions. What is the maximum number of one-operand instructions that can be encoded using the fixed 16-bit instruction length?

Exercise 20

A digital computer has a memory unit with 32 bits per word. The instruction set consists of 110 different operations. All instructions have an operation code part (opcode) and two address fields: one for a memory address and one for a register address. This particular system includes eight general purpose, user addressable registers. Registers may be loaded directly from the memory, and memory may be uploaded directly from the registers. Direct memory-to-memory data movement operations are not supported. Each instruction is stored in one word of memory.

1. How many bits are needed for the opcode?
2. How many bits are needed to specify the register?
3. How many bits are left for the memory address part of the instruction?
4. What is the maximum allowable size for memory?

Exercise 21

For the code sequence below, state whether it must stall, can avoid stalls using only forwarding, or can execute without stalling or forwarding. Consider the 5-stages pipeline.

```
addi $t1, $t0, 1
addi $t2, $t0, 2
addi $t3, $t0, 2
addi $t3, $t0, 4
addi $t5, $t0, 5
```

Exercise 22

Assume 1% of the runtime of a program is not parallelizable. This program is run on 61 cores of an Intel Xeon Phi. Under the assumption that the program runs at the same speed on all of those cores, and there are no additional overheads, what is the parallel speedup?

Exercise 23

You have some protein matching code. It completes in 200 hours on your current machine and spends 20% of the time doing integer operations. How much faster must you make the integer unit to make the code run 10 hours faster?

Exercise 24

Supposing that you have a computer with 1 MHz CPU clock rate is executing a program taking 45 million cycles. What's the CPU time?

Exercise 25

A compiler designer is trying to decide between two code sequences for a particular machine. Based on the hardware implementation, there are three different classes of instructions: Class A, Class B, and Class C, and they require one, two, and three cycles respectively).

1. The first code sequence has 5 instructions: 2 of A, 1 of B, and 2 of C.
2. The second sequence has 6 instructions: 4 of A, 1 of B, and 1 of C.
3. Which sequence will be faster? How much? What is the CPI for each sequence?

Exercise 26

Suppose we have two implementations of the same instruction set architecture (ISA). For some program, Machine A has a clock cycle time of 10 ns. and a CPI of 2.0 Machine B has a clock cycle time of 20 ns. and a CPI of 1.2. Which machine is faster for this program, and by how much?

Exercise 27

Supposing you have a processor with clock equals 400MHz. The pipeline of this processor has 6 levels stages). What is the timed needed to execute 1000 instructions, a) if the processor uses the pipeline, b) without the pipeline usage?

Exercise 28

What is the overall speedup if you make 10% of a program 90 times faster?

Exercise 29

What is the overall speedup if you make 90% of a program 10 times faster?

Exercise 30

We are considering an enhancement to the processor of a web server. The new CPU is 20 times faster on search queries than the old processor. The old processor is busy with search queries 70% of the time, what is the speedup gained by integrating the enhanced CPU?

Exercise 31

We are considering an enhancement to the processor of a server. The new CPU 10X faster. I/O bound server, so 60% time waiting for I/O. What is the speedup gained by integrating the enhanced CPU?

Exercise 32

Suppose a cache memory is 5 times faster than main memory and suppose that the cache can be used over 90% of the time. How much speed-up can be gained by using the cache?

Exercise 33

You have some graph processing code which takes four days to execute on your current machine. Of that time 20% of the time is spent performing integer operations, and 35% of the time is spent performing I/O operations.
Which of the following is the better tradeoff?

1. A compiler optimization that reduces the number of integer instructions by 25% (assume that each integer operation still takes the same amount of time).
2. A hardware optimization that reduces the latency of each I/O operation from 6μs to 5μs.

Exercise 34

Assume that we have an N-way set associative cache that has a block size of 8 bytes and the number of sets is 256. Assume that the replacement policy is least recently used (LRU).

1. If the capacity of the cache is 8192 bytes, what is then the associativity of the cache?
2. Sketch the layout of the cache content for the cache if we assume that N = 2. The figure should include the ways and columns for the valid bit, the tag, and the data. What is then the capacity of the cache?

Exercise 35

Each of the following statements pertains to the miss rate of caches. Mark each statement as true or false. Briefly explain your reasoning; present a counterexample if the statement is false.

1. A two-way set associative cache always has a lower miss rate than a direct mapped cache with the same block size and total capacity.
2. A 16-KB direct mapped cache always has a lower miss rate than an 8-KB direct mapped cache with the same block size.
3. An instruction cache with a 32-byte block size usually has a lower miss rate than an instruction cache with an 8-byte block size, given the same degree of associativity and total capacity.

Exercise 36

If a computer uses 64-bit virtual addresses, how much virtual memory can it access? Note that 2^{40} bytes = 1 terabyte, 2^{50} bytes = 1 petabyte, and 2^{60} bytes = 1 exabyte.

Exercise 37

Assume that you have a virtual memory, where the page size is 4096 bytes. The virtual memory address is 32-bits, and the physical memory address is 18-bits, meaning that 2^{18} = 26244 bytes of data can be stored in the main memory.

1. How many bits does the page offset consist of?
2. How many physical pages exist?
3. How many virtual pages exist?
4. Where is the mapping between virtual page numbers and physical page numbers stored?
5. Who or what is responsible for updating this mapping?
6. Who or what is responsible for translating between virtual page numbers and physical page numbers?

Exercise 38

A computer system uses 16-bit memory addresses. It has a 2K-byte cache organized in a direct-mapped manner with 64 bytes per cache block. Assume that the size of each memory word is 1 byte.

1. Calculate the number of bits in each of the Tag, Block, and Word fields of the memory address.
2. When a program is executed, the processor reads data sequentially from the following word addresses: 128, 144, 2176, 2180, 128, 2176 All the above addresses are shown in decimal values. Assume that the cache is initially empty. For each of the above addresses, indicate whether the cache access will result in a hit or a miss.

Exercise 39

A block-set associative cache memory consists of 128 blocks divided into four block sets. The main memory consists of 16,384 blocks and each block contains 256 eight-bit words.

1. How many bits are required for addressing the main memory?
2. How many bits are needed to represent the TAG, SET and WORD fields?

Exercise 40

A 4-way set associative cache memory unit with a capacity of 16 KB is built using a block size of 8 words. The word length is 32 bits. The size of the physical address space is 4 GB. The number of bits for the TAG field is .....

Exercise 41

Consider a direct mapped cache with 8 cache blocks (0-7). If the memory block requests are in the order:

3, 5, 2, 8, 0, 6, 3, 9, 16, 20, 17, 25, 18, 30, 24, 2, 63, 5, 82, 17, 24

Which of the following memory blocks will not be in the cache at the end of the sequence?

1. 3
2. 18
3. 20
4. 30

Also, calculate the hit ratio and miss ratio.

Exercise 42

A hierarchical memory system has the following specification, 20 MB main storage with word size equals 4 bytes, page size 8 words. What will be the hit ratio if the page address trace of a program has the pattern 0, 1, 2, 3, 0, 1, 2, 4 following LRU page replacement technique?

Exercise 43

Consider an array A [100] and each element occupies 4 words. A 32-word cache is used and divided into 8-word blocks. What is the hit ratio for the following code-

```
for (i=0; i < 100; i++)
    A[i] = A[i] + 10;
```

Tutorial exercises Solutions

Exercise 0 Solution:

Part 1)

The binary equivalent of $(D3E.A6)_{16}$ is: **1101 0011 1110.1010 0110** (binary)

The hexadecimal equivalent of $(1011001110.011011101)_2$ is: **2CE.6E8** (hexadecimal)

Part 2)

The octal equivalent of $(BD.C4)_{16}$ is: 575.610 (octal)

The hexadecimal equivalent of $(565.013)_8$ is: 175.058 (hexadecimal)

Exercise 1 Solution:

A load-store architecture is one in which only load and store instructions can access the memory system. In other GPR architectures, some or all of the other instructions may read their operands from or write their results to the memory system. The primary advantage of non-load-store architectures is the reduced number of instructions required to implement a program and lower pressure on the register file. The advantage of load- store architectures is that limiting the set of instructions that can access the memory system makes the micro- architecture simpler, which often allows implementation at a higher clock rate. Depending on whether the clock rate increase or the decrease in number of instructions is more significant, either approach can result in greater performance.

Exercise 2 Solution:

A large register file and a large data cache both serve the purpose of reducing memory traffics. From an implementation point of view, the same chip area can be used for either a large register file or a large data cache. With a larger register set, the instruction set must be changed so that it can address more register in the instructions. From a programming point of view, registers can be manipulated by program code, but the data cache is transparent to the user. In fact, the data cache is primarily involved in load/store operations. The addressing of a cache involves address translation and is more complicated than that of a register file.

Summary of the Trade-Offs:

Consideration	Large Register File	Large Data Cache
Speed	Fastest access (near-zero latency).	Slower than registers due to potential cache misses.
Size	Small size (limited by hardware constraints).	Larger size, holds more data, can store more of the working set.

Consideration	Large Register File	Large Data Cache
Cost	More expensive to implement.	More expensive for larger caches, but scalable (L1, L2, L3).
Power	Higher power consumption due to faster switching.	Power consumption can be managed with hierarchy but still significant.
Access Pattern	Ideal for small, frequently used data (registers for computation).	Better for larger, less frequently accessed data (cache).
Scalability	Difficult to scale for very large files.	Easier to scale and organize in a hierarchical manner (L1, L2, L3).
Software Impact	Requires careful management of register usage by the compiler.	Cache management handled more by hardware, benefits from locality optimizations.

Exercise 3 Solution:

Difference Between SoC and SiP

Aspect	**System on Chip (SoC)**	**System in Package (SiP)**
Definition	All components integrated into **one chip**	Multiple separate chips housed in **one package**
Components Integrated	CPU, GPU, RAM, Neural Engine, I/O, Security	Multiple chips (e.g., CPU, RAM, power management)
Power Efficiency	More efficient due to higher integration	Slightly less efficient due to separate chips
Size	Smaller footprint, compact design	Larger footprint but allows for flexible configurations
Latency	Lower latency due to direct integration	Potentially higher latency due to separate components
Flexibility	Less flexibility, all components are fixed	More flexibility as individual chips can be chosen and replaced
Cost	Higher integration cost but lower overall system cost	Flexibility increases cost of packaging and integration
Use Case	Mobile devices, wearables, laptops (e.g., iPhone, iPad, M1 Macs)	IoT devices, wearables (e.g., Apple Watch), specialized applications

Comparison: Apple A-series/M-series vs. Intel Processors

Feature	**Apple A-series/M-series**	**Intel Processors**
Architecture	ARM-based (Custom Apple cores)	x86 (Traditional Intel architecture)
Target Devices	iPhone, iPad, Mac (M1, M2), Apple Watch	Laptops, Desktops, Servers, Workstations
Core Configuration	High-performance cores (e.g., Avalanche, Firestorm) & Energy-efficient cores (e.g., Icestorm)	Varies by series: Core i3, i5, i7, i9 with multiple cores (typically 4 to 18 cores)
Manufacturing Process	5nm (A-series: A14, A15; M1, M2)	10nm (Intel 10nm SuperFin, Alder Lake)

CPU Performance	Extremely high performance with efficient power use (due to ARM's design)	High performance with focus on multitasking and legacy x86 apps
Graphics (GPU)	Custom Apple-designed GPUs (e.g., 8- core, 16-core)	Integrated Intel Iris Xe graphics (i3, i5, i7) or discrete GPUs (i9, for gaming or professional use)
AI/ML Support	Dedicated Neural Engine for AI and machine learning	Intel AI and Deep Learning capabilities (via OpenVINO, etc.)
Unified Memory Architecture	Yes, Unified Memory (CPU, GPU, and Neural Engine share memory)	No, CPU, GPU, and other components have separate memory (except for integrated graphics)
Power Efficiency	Very power-efficient, especially with ARM architecture (low power consumption in M-series)	Power consumption depends on cores; higher power for higher performance (especially in desktop CPUs)
Heat & Thermal Management	Excellent thermal efficiency; low heat output (e.g., M1 MacBook Air has no fan)	Higher heat output, often requiring active cooling (e.g., fans in laptops)
Performance per Watt	High performance per watt due to ARM's custom design and efficient cores	Performance varies, but typically lower performance per watt in high-end CPUs
Security Features	Secure Enclave (hardware-based security)	Intel's Hardware-based Security (e.g., Intel SGX, TPM)
Integrated Components	SoC (System on Chip) integrates CPU, GPU, Neural Engine, RAM, I/O, etc.	Typically separate chips (CPU, GPU, RAM, I/O), though some integrate GPU in lower-end chips (Intel Core)
OS Compatibility	macOS, iOS, iPadOS	Windows, Linux, macOS (with Boot Camp or virtual machines)
Software Ecosystem	Optimized for Apple ecosystem (macOS, iOS, iPadOS, etc.)	Compatible with a wide variety of software for Windows, Linux, and macOS (Intel-based Macs)
Upgradability	Fixed, cannot upgrade CPU, RAM, or other components (e.g., M1 Macs)	CPUs can be upgraded (for desktops), but often requires motherboard replacement
Virtualization & Emulation	Supports virtualization via macOS or iOS, Rosetta 2 for Intel app emulation	Full virtualization support, including Docker, Hyper-V, VMware, etc.
Market Segment	Mobile devices, ultrathin laptops, premium desktops (Mac)	Laptops, desktops, workstations, and servers (wide range from budget to high-end)
Feature	Apple A-series/M-series	Intel Processors
Performance in Multitasking	Optimized for efficient multitasking with ARM's design (power-efficient cores handle background tasks)	Excellent multitasking, especially with high core-count CPUs in i7/i9 series for multi-threaded tasks

Exercise 4 Solution

- **First Generation Computers used:**
  a) Transistors b) Vacuum tubes c) Integrated circuits d) Microprocessors
- **Which of the following is NOT a function of a computer?**
  a) Input b) Processing c) Output d) Communication with other systems
- **Which of the following best describes the Harvard Architecture?**
  a) Single memory for both data and instructions
  b) Separated memory for data and instructions, allowing parallel processing
  c) A single CPU with no memory
  d) Memory that uses magnetic tapes for storage
- **What is the primary function of the Arithmetic Logic Unit (ALU) in a processor?**
  a) Fetch instructions from memory
  b) Perform mathematical and logical operations
  c) Control the flow of data between the CPU and memory
  d) Store instructions temporarily for fast access
- **What type of memory is used for the temporary storage of data while the computer is powered on?**
  a) ROM b) RAM c) Cache memory d) Secondary memory
- **Which of the following is the main advantage of using microprocessors in computers?**
  a) Increased reliability and reduced size b) Increased complexity
  c) Greater need for cooling d) Reduced speed of processing
- **What is the main difference between primary and secondary memory?**
  a) Secondary memory is temporary, while primary memory is permanent.
  b) Primary memory is faster, while secondary memory is slower.
  c) Secondary memory is volatile, while primary memory is non-volatile.
  d) Primary memory is used for input and output, while secondary memory is for processing.
- **Which of the following components is responsible for directing operations within the CPU?**
  a) Control Unit (CU) b) Arithmetic Logic Unit (ALU)
  c) Memory d) Registers
- **Which architecture uses separate memory for instructions and data?**
  a) Von Neumann Architecture
  b) Harvard Architecture
  c) RISC Architecture
  d) CISC Architecture
- **The superscalar approach can be used on..... architecture.**
  A. RISC **B.** CISC **C.** neither RISC nor CISC **D.** both RISC and CISC
- **The essence of the ..... approach is the ability to execute instructions independently and concurrently in different pipelines.**
  A. scalar **B.** branch **C.** superscalar **D.** flow dependency
- **Which of the following is a fundamental limitation to parallelism with which the system must cope?**
  A. procedural dependency **B.** resource conflicts **C.** anti-dependency **D.** all of the above
- **The situation where the second instruction needs data produced by the first instruction to execute is referred to as ..... .**
  A. true data dependency **B.** output dependency **C.** procedural dependency **D.** anti-

dependency

- **The instructions following a branch have a ..... on the branch and cannot be** executed until the branch is **executed**.

A. resource dependency **B.** procedural dependency **C.** output dependency **D.** true data dependency

Exercise 5 Solution:

1. **MOV R1, #5 ⟶ Immediate Addressing**
2. **ADD R2, [R3] ⟶ Register Indirect Addressing**
3. **LDR R1, [R2 + 4] ⟶ Indexed Addressing**
4. **MOV R1, [0x2000] ⟶ direct Addressing**

Exercise 6 Solution

1. What is the operand here? **#50**
2. What addressing mode is being used? **Immediate Addressing**
3. What is the value of register R1 after execution? **50**

Exercise 7 Solution

1. What are the operands for this instruction? **R2, R3**
2. What addressing mode is being used? **Register addressing**
3. What is the result of the addition? **R1=R2+R3**

Exercise 8 Solution

1. What is the operand here? **[2000]**
2. What addressing mode is being used? **Direct addressing**
3. What value is loaded into register R1? **Content of memory address 2000**

Exercise 9 Solution

1. What is the operand here? **[R2]**
2. What addressing mode is being used? **Indirect addressing**
3. If the value in R2 is 1000, what is the effective address? **1000**

Exercise 10 Solution

1. What is the effective address? **R2+5**
2. What addressing mode is being used? **Indexed Addressing**

Exercise 11 Solution

1. What is the base address? **R2**
2. What is the offset? **100**
3. What addressing mode is being used? **Indexed addressing**

Exercise 12 Solution

1. What is the effective address of the jump target? **PC+200**
2. What addressing mode is being used? **PC Relative addressing**

Exercise 13 Solution

1. What is the effective address? **R2 + 8**
2. What addressing mode is being used? **Indexed Addressing**

Exercise 14 Solution

1. Where is the operand stored? **R1**
2. What addressing mode is being used? **Stack addressing**

Exercise 15 Solution:

1. **RAW (Read After Write) Dependency:**

I2 depends on I1: I2 reads the value of R1 that was written by I1.

WAW (Write After Write) Dependency:

I5 depends on I2: I5 writes to R1, which is the same register that was written by I2.

2. **Rename** the registers from part (a) to prevent dependency problems. Identify references to initial register values using the subscript "a" to the register reference.

=> **The renamed instructions are as follows:**

I1: Ra1 = 100

I2: Ra2 = Ra3 + Ra4

I3: Ra3 = Ra4 - 25

I4: Ra4 = Ra1 + Ra5

I5: Ra6 = Ra2 + 30

Exercise 16 Solution:

a) WAW: 1,3; RAW: 1,2; 1,3 ; 1,4 ; 3,4; 2,5; 4,5 WAR: 2,3;

b) A hazard is created when there is a dependency between instructions and they are close enough that the overlap caused by pipelining (or reordering) would change the order of access to the dependent operand.

c) In a true dependency information is transmitted between instructions, while this is not the case for name dependencies.

d) Name dependencies: WAR, WAW True dependencies: RAW

Exercise 17 Solution:

The second instruction is dependent upon the first (because of R2)

The third instruction is dependent upon the first (because of R2)

The fourth instruction is dependent upon the first (because of R2)

The fourth instruction is dependent upon the second (because of R4)

All these dependencies can be solved by forwarding

Exercise 18 Solution:

Registers R11 and R12 are read during the fifth clock cycle.

Register R1 is written at the end of fifth clock cycle.

Exercise 19 Solution:

2^{10}= 1024

Exercise 20 Solution:

1- $\log_2(110) \approx 6.8 \Rightarrow \lceil 6.8 \rceil = 7$ bits. So, **7 bits** are needed for the opcode.
2- $\log 2(8) = 3 \Rightarrow$ 3 bits. So, **3 bits** are needed to specify the register.
3- bits for memory address=32−7−3=22 bits. So, **22 bits** are left for the memory address part of the instruction.
4- max memory size=2^{22} words.

Exercise 21 Solution:

No stalling or forwarding is required.

Exercise 22 Solution:

$$Speedup = \frac{1}{0{,}01+\frac{(0{,}99)}{61}} = 38{,}125$$

Exercise 23 Solution:

$$Speedup = \frac{200}{190} = \frac{1}{(1-x)+\frac{x}{S_{int}}} = \frac{1}{(1-0{,}2)+\frac{0{,}2}{S_{int}}}$$

$$\frac{190}{200} = \frac{0{,}2}{S_{int}} + (1 - 0{,}2)$$

$$\frac{190}{200} - 0{,}8 = \frac{0{,}2}{S_{int}}$$

$$S_{int} = \frac{0{,}2}{\frac{190}{200} - 0{,}8} = 1{,}33$$

Exercise 24 Solution:

45,000,000 * (1 / 1,000,000)= 45 seconds

Exercise 25 Solution:

# of cycles for first code = (2 * 1) + (1 * 2) + (2 * 3) = 10 cycles

# of cycles for second code = (4 * 1) + (1 * 2) + (1 * 3) = 9 cycles

CPI for first code = 10 / 5 = 2

CPI for second code = 9 / 6 = 1.5

10 / 9 = 1.11 times

Exercise 26 Solution:

Assume that # of instructions in the program is 1,000,000,000.

CPU TimeA= 10^9* 2.0 * 10 * 10^{-9} = 20 seconds

CPU TimeB= 10^9* 1.2 * 20 * 10^{-9}= 24 seconds

Machine A is faster 24/20= 1.2 times

Exercise 27 Solution:

Total execution time for **m** instructions of **n** sub-instructions in sequential mode is: $T_t = n * m * T_e$

In pipeline mode (pipeline of n stages): $T_t = (n + m - 1) * T_e$

T_e is the time to execute one sub-instruction (cycle). In our case, **n=6**, **m=1000**

T_e= 1/400*10-6= 2.5 ns

T_t = 1000*6*2.5ns=15μs **(without pipeline usage)**

T_t = (6+999)*2.5ns=2512.5ns **(using pipeline)**

Exercise 28 Solution:

$$Speedup = \frac{1}{(1-f)+\frac{f}{s}} = \frac{1}{(1-0,1)+\frac{0,1}{90}} = 1,11$$

Exercise 29 Solution:

$$Speedup = \frac{1}{(1-f)+\frac{f}{s}} = \frac{1}{(1-0,9)+\frac{0,9}{10}} = 5,26$$

Exercise 30 Solution

Solution:

$$Speedup = \frac{1}{(1-f)+\frac{f}{s}} = \frac{1}{(1-0,7)+\frac{0,7}{20}} = 2,985$$

Exercise 31 Solution:

$$Speedup = \frac{1}{(1-f)+\frac{f}{s}} = \frac{1}{(1-0,4)+\frac{0,4}{10}} = 1,56$$

Exercise 32 Solution

$$\boldsymbol{speed\ up = \frac{1}{(1-p)+\frac{p}{N}} = \frac{1}{(1-0.9)+\frac{0.9}{5}} = 3.57}$$

Exercise 33 solution

Total execution time: 4 days = 4×24×60×60=345600 seconds

Time spent on Integer ops = 20% of time = 0.20×345600=69120 seconds

Time spent on I/O ops = 35% of time = 0.35×345600=120960 seconds

Time spent on the rest of operations: 345600−69120−120960=155520 seconds

1. **Evaluating option (a):**

   New time spent on Integer ops = 69120 × (1−0.25) = 51840 seconds

   New total time= 120960 + 51840 +155520= 328320 seconds

   Speed up= $\boldsymbol{\frac{\text{original time}}{New\ total\ time} = \frac{345600}{328320} = 1.053}$

2. **Evaluating option (b):**

   Enhancement factor of I/O: N=6/5=1.2

   $$\boldsymbol{speed\ up = \frac{1}{(1-p)+\frac{p}{N}} = \frac{1}{(1-0.35)+\frac{0.35}{1.2}} = 1.062}$$

3. **Therefore, option (b) gave a better speedup.**

Exercise 34 solution

Cache size=Number of sets × Associativity × Block size

Given that:

- Cache size = 8192 bytes
- Number of sets = 256
- Block size = 8 bytes

Let n be the associativity:

8192=256× n ×8 => n=$\boldsymbol{\frac{8192}{256*8} = 4}$

Noting that: "**4-way set-associative cache with 256 sets**" **means**: You have **256 sets**, each set has **4 blocks (or lines)**.

a) The associativity of the cache is **4** ⟶ It is a **4-way set associative** cache.

b) Given:

- **Block size** = 8 bytes
- Number of sets = **256**

- Associativity = **2 (so, 2 blocks per set)**

The cache sketch:

Valid bit: indicating that this info is useful; **Dirty bit**: indicating if it has changed value -writable

Valid bit	Tag	Data block (8 bytes)	Way 0	Set 0
Valid bit	Tag	Data block (8 bytes)	Way 1	
Valid bit	Tag	Data block (8 bytes)	Way 0	Set 1
Valid bit	Tag	Data block (8 bytes)	Way 1	
.....	.....	.....	.....	....
Valid bit	Tag	Data block (8 bytes)	Way 0	Set 255
Valid bit	Tag	Data block (8 bytes)	Way 1	

Total capacity=Number of sets × Ways × Block size=256 × 2 × 8 = 4096 bytes

Exercise 35 solution

a) **False, not always.** For instance, in **sequential access patterns** (where each block has its own unique cache line), the two-way set-associative cache might introduce **overhead** because not all sets are fully utilized.

b) **False, not always**. Imagine a small program working with a data set of just 4 KB. An **8-KB cache** can store this data perfectly, so there would be **few misses**. In contrast, the **16-KB cache** might have to store unnecessary data, causing more **conflict misses**.

c) **False** — Smaller block sizes are typically more efficient for instruction caches.

Exercise 36 solution

2^{64} bytes=**16 exabytes**. The computer can access up to **16 exabytes (EB)** of virtual memory.

Exercise 37 solution

Page size: 4096 bytes (i.e., $\mathbf{2^{12}}$ bytes).

Virtual memory address: 32 bits.

Physical memory address: 18 bits.

Main memory size: $\mathbf{2^{18}}$ bytes (i.e., 262144 bytes).

1. The **page offset** refers to the part of the address that specifies the exact location within a page. The page offset consists of **12 bits**.
2. Number of physical pages= $\frac{\textbf{Physical memory size}}{\textbf{Page size}} = \frac{2^{18}}{2^{12}} = \mathbf{2^6}$

   There are **64 physical pages**.
3. Number of virtual pages= $\frac{\textbf{Virtual memory size}}{\textbf{Page size}} = \frac{2^{32}}{2^{12}} = \mathbf{2^{20}}$

   There are **1048576 virtual pages**

4. The mapping is stored in the **page table**, which resides in **main memory**.
5. The **operating system (OS)** is responsible for updating the mapping.
6. The Memory Management Unit (MMU) is responsible for translating between virtual page numbers and physical page numbers. The Memory Management Unit (MMU) is responsible for translating between virtual page numbers and physical page numbers.

Exercise 38 solution

Memory address size: 16 bits.

Cache size: 2 KB (2 * 1024 bytes = 2048 bytes).

Block size: 64 bytes per cache block.

Memory word size: 1 byte.

Cache organization: Direct-mapped.

Word field:

- **Block size** is 64 bytes, the **word field** is the part of the address that specifies the offset within a cache block.
- 64 bytes= $\mathbf{2^6}$ bytes. Therefore, the **word field** will require **6 bits** to address each byte within the block.

Block field:

- The total cache size is **2 KB** (2048 bytes), and the cache is direct-mapped.
- Each cache block is **64 bytes**, and the cache is **direct-mapped**, meaning each memory block maps to exactly one cache line. The number of cache lines is: $\frac{\textbf{Cache size}}{\textbf{Block size}} = \frac{2^{11}}{2^6} = 2^5 = 32\ \boldsymbol{Cache\ lines}$

The **block field** in the memory address specifies which cache line to map to. Since there are 32 cache lines, you need $\log_2(32)=5$ bits to represent the block number (i.e., which cache line to use).

Tag field:

- The **tag field** is the remaining part of the memory address after removing the block and word fields.
- The total address size is **16 bits**. Thus, the **tag field** will occupy:

Tag field size=16−6−5=5 bits

Address (Decimal)	Address (Binary)	Tag (5 bits)	Block Index (5 bits)	Word Offset (6 bits)	Hit/Miss
128	0000000010000000	00000	00010	000000	Miss
144	0000000010010000	00000	00010	010000	Hit
2176	0000100010000000	00001	00010	000000	Miss
2180	0000100010000100	00001	00010	000100	Hit
128	0000000010000000	00000	00010	000000	Miss
2176	0000100010000000	00001	00010	000000	Miss

Exercise 39 solution

22 bits are required to address the main memory; number of bits for the main memory = $\log_2(16384)+\log_2(256)$

The bit breakdown is:

- **TAG field**: 9 bits= 22- (5+8)
- **SET field**: 5 bits = log_2 (number of sets in cache)= $log_2\left(\frac{Total\ blocks\ in\ cache}{number\ of\ block\ sets}\right) = log_2(\frac{128}{4})$
- **WORD field**: 8 bits= $log_2(256)$

Exercise 40 solution

Cache capacity: 16 KB.

Block size: 8 words per block.

Word length: 32 bits (4 bytes per word).

Physical address space: 4 GB (4 * 1024 * 1024 * 1024 bytes)=2^{32}.

- Number of blocks in the cache= $\frac{Cache\ Capacity}{Block\ size} = \frac{16KB}{32\ Bytes} = 512\ blocks$
- Number of sets in the cache= $\frac{Number\ of\ blocks}{Associativity} = \frac{512}{4} = 128\ sets$
- Number of bits for word offset= $log_2(32) = 5$
- Number of bits for Set index= $log_2(128) = 7$
- TAG bits=Total address bits−(Set index bits+Block offset bits) =32−(7+5)=32−12=20bits.

Exercise 41 solution

Cache type: Direct-mapped cache with **8 cache blocks** (numbered 0-7).

Memory block requests: 3, 5, 2, 8, 0, 6, 3, 9, 16, 20, 17, 25, 18, 30, 24, 2, 63, 5, 82, 17, 24.

In a **direct-mapped** cache, each memory block maps to exactly one cache block. The mapping is typically determined by the following formula:

Cache block=(Memory block mod Number of cache blocks))

Since we have 8 cache blocks (0-7), the cache block for a memory block will be determined by:

Cache block=Memory block mod 8

Memory Block Access Sequence:

memory block	cache block index (mod 8)	cache state
3	3mod8=3	[_, _, _, 3, _, _, _, _]
5	5mod8=5	[_, _, _, 3, _, 5, _, _]
2	2mod8=2	[_, _, 2, 3, _, 5, _, _]
8	8mod8=0	[8, _, 2, 3, _, 5, _, _]
0	0mod8=0	[0, _, 2, 3, _, 5, _, _]
6	6mod8=6	[0, _, 2, 3, _, 5, 6, _]
3	3mod8=3	[0, _, 2, 3, _, 5, 6, _] (Hit)
9	9mod8=1	[0, 9, 2, 3, _, 5, 6, _]
16	16mod8=0	[16, 9, 2, 3, _, 5, 6, _]
20	20mod8=4	[16, 9, 2, 3, 20, 5, 6, _]

17	17mod8=1	[16, 17, 2, 3, 20, 5, 6, _]
25	25mod8=1	[16, 25, 2, 3, 20, 5, 6, _]
18	18mod8=2	[16, 25, 18, 3, 20, 5, 6, _]
30	30mod8=6	[16, 25, 18, 3, 20, 5, 30, _]
24	24mod8=0	[24, 25, 18, 3, 20, 5, 30, _]
2	2mod8=2	[24, 25, 2, 3, 20, 5, 30, _]
63	63mod8=7	[24, 25, 2, 3, 20, 5, 30, 63]
5	5mod8=5	[24, 25, 2, 3, 20, 5, 30, 63] (Hit)
82	82mod8=2	[24, 25, 82, 3, 20, 5, 30, 63]
17	17mod8=1	[24, 17, 82, 3, 20, 5, 30, 63]
24	24mod8=0	[24, 17, 82, 3, 20, 5, 30, 63] (Hit)

Block 18 not found in cache at the end of sequence, hit ratio (3/21), miss ratio (18/21)

Exercise 42 solution

Number of cache blocks= Cache size/ Page size=256/32=8.

Page Address	Cache state after access	Hit/Miss
0	[0, , , , , , ,]	Miss
1	[0, 1, , , , , ,]	Miss
2	[0, 1, 2, , , , ,]	Miss
3	[0, 1, 2, 3, , , ,]	Miss
0	[0, 1, 2, 3, , , ,]	Hit
1	[0, 1, 2, 3, , , ,]	Hit
2	[0, 1, 2, 3, , , ,]	Hit
4	[0, 1, 2, 3,4, , ,]	Miss

Hits : 3 (for pages 0, 1, and 2 during their second access)

Misses: 5 (for pages 0, 1, 2, 3, and 4 during their first access)

Total accesses=8

Hit ratio: 3/8 Miss ratio: 5/8

Exercise 43 Solution:

Number of lines in cache= Cache size / Line size= 32 words / 8 words= 4 lines

Since each element of the array occupies 4 words, so-

Number of elements that can be placed in one line = 2

Now, let us analyze the statement-

A[i] = A[i] + 10;

For each i,

- Firstly, the value of A[i] is read.

- Secondly, 10 is added to the A[i].
- Thirdly, the updated value of A[i] is written back.

Thus,

- For each i, A[i] is accessed two times.
- Two memory accesses are required-one for read operation and other for write operation.

Assume the cache is all empty initially.

In the first loop iteration,

- First, value of A[0] has to be read.
- Since cache is empty, so A[0] is brought in the cache.
- Along with A[0], element A[1] also enters the cache since each block can hold 2 elements of the array.

Cache Memory

- Thus, For A[0], a miss occurred for the read operation.
- Now, 10 is added to the value of A[0].
- Now, the updated value has to be written back to A[0].
- Again cache memory is accessed to get A[0] for writing its updated value.
- This time, for A[0], a hit occurs for the write operation.

In the second loop iteration,

- First, value of A[1] has to be read.
- A[1] is already present in the cache.
- Thus, For A[1]. a hit occurs for the read operation.
- Now, 10 is added to the value of A[1].
- Now, the updated value has to be written back to A[1].
- Again cache memory is accessed to get A[1] for writing its updated value.
- Again, for A[1], a hit occurs for the write operation.

In the similar manner, for every next two consecutive elements-

- There will be a miss for the read operation for the first element.
- There will be a hit for the write operation for the first element.
- There will be a hit for both read and write operations for the second element.

Likewise, for 100 elements, we will have 50 such pairs in cache and in every pair, there will be one miss and three hits.

Thus,

- Total number of hits = 50 x 3 = 150

- Total number of misses = 50 x 1 = 50
- Total number of references = 200 (100 for read and 100 for write)

Thus,

- Hit ratio = 150 / 200 = 3 / 4 = 0.75
- Miss ratio = 50 / 200 = 1 / 4 = 0.25

Labs

Exercise 1

Write the NASM program that does the following:

```
Function Sum (integer: a,b) : integer
        { a=a+b}
```

Exercise 2

Write the NASM program that calculates the factorial of an integer.

Exercise 3

Write the NASM program which calculates the sum of n integers (function Sum +Loop). Parameter passing must be done by register. Do the same by passing the parameters using the stacks.

Exercise 4

Write the NASM program that does the permutation of two variables (swapping). Do not use the instruction exchange for the swapping task.

Exercise 5

Write the NASM program that does the following:

```
Function A (float: a, b) : float
        {
                if (a>b)
                   a = a-b;
                else
                   a= b-a ;
        }
```

Exercise 6

Using the following command (Linux)

```
gcc -S -g file.c -o file.asm
```

Generate the Assembly program, comment it and execute it from the following c program:

```
int divide (int a, int b){
```

```
return a/b ;
}
int main(void)
{
Printf ("result of d= %d", divide (7,3));
return 0;
}
```

Exercise 7

Write the NASM program which calculates the Fibonacci sequence, recalling that: $F_0=0$, $F_1=1$ for n>1, $Fn=F_{n-1}+F_{n-2}$

Exercise 8

Write the NASM program(function) that calculates the sum of an array of **n** integers.

Exercise 9

Write the NASM program(function) that calculates the sum of a matrix of n*m integers.

Labs solutions

Note: To solve the exercises, we use NASM on Linux system, if you do not have it installed, you may install Linux virtual machine or install WLS on your windows machine.

Exercise 1 Solution

Passing Function parameters using stack	**Passing Function parameters using Registers**

global _start
section .text
_start: push qword [a]
push qword [b]
call Sum_a
xchg eax, ebx ; value for exit()
mov eax, 1 ; exit
int 0x80 ; exit(...)
Sum_a: push rbp
mov rbp, rsp
mov rax, [rbp+16]
mov rbx, [rbp+24]
add rax, rbx
pop rbp
ret 16
section .data
a dq 1
b dq 3
	global _start
	;----------
	section .data
	a dq 1
	b dq 3
	;----------
	section .text
	_start:
	mov rax, [a]
	mov rbx, [b]
	call Sum_a
	;mov rax, rcx
	xchg eax, ebx ; value for exit()
	mov eax, 1 ; exit
	int 0x80 ; exit(...)
	Sum_a:
	add rax, rbx ; rax=rax+rbx
	ret

Exercise 2 Solution

```
global _start
section .data
n dq 3
section .text
_start:
        push qword [n]
        call fact
        xchg eax, ebx ; value for exit()
        mov eax, 1 ; exit
        int 0x80 ; exit(...)
fact:  push rbp
        mov rbp, rsp
        mov rax,  [rbp+16]
        cmp rax, 1
        jg p1
```

```
            mov rax, 1
            jmp fin_
p1:  sub rax, 1
            push rax
            call fact
            mov rbx,  [rbp+16]
            imul rax, rbx
fin_:  pop rbp
            ret 8
```

Exercise 3 Solution

Passing Function parameters using registers	Passing Function parameters using stack

global _start
section .data
n dq 3
section .text
_start: mov r11, [n]
call sumN
xchg eax, ebx ; value for exit()
mov eax, 1 ; exit
int 0x80 ; exit(...)
sumN: mov r12, 0
loop: cmp r11, 0
jg next_
jmp End_
next_: add r12, r11
sub r11, 1
jmp loop
End_: ret
	global _start
	section .data
	n dq 3
	section .text
	_start:
	push qword [n]
	call sumN
	xchg eax, ebx ; value for exit()
	mov eax, 1 ; exit
	int 0x80 ; exit(...)
	sumN: push rbp
	mov rbp, rsp
	mov r11, [rbp+16]
	mov r12, 0
	loop: cmp r11, 0
	jg next_
	jmp End_
	next_: add r12, r11
	sub r11, 1
	jmp loop
	End_: pop rbp
	ret 8

Exercise 4 Solution

```
global _start
section .text
_start:
            mov r11, [a]
            mov r12, [b]
```

```
        mov [a], r12
        mov [b], r11
        xchg eax, ebx ; value for exit()
        mov     eax, 1 ; exit
        int     0x80  ; exit(...)
section .data
a dq 12
b dq 25
```

Exercise 5 Solution

```
global _start
section .data
a dq 3
b dq 9
section .text
_start: mov r11, [a]
        mov r12, [b]
        call Fun_A
        xchg eax, ebx ; value for exit()
        mov     eax, 1 ; exit
        int     0x80  ; exit(...)
Fun_A: cmp r11, r12
        jg if_part
        sub r12, r11
        mov r11, r12
        jmp end_
if_part: sub r11, r12
end_: mov [a], r11
        ret
```

Exercise 6 Solution

```
;gcc -S -g exo7.c -o exo7-r.asm
diviser(int, int):
push rbp
mov rbp, rsp
mov DWORD PTR [rbp-4], edi
mov DWORD PTR [rbp-8], esi
mov eax, DWORD PTR [rbp-4]
cdq
idiv DWORD PTR [rbp-8]
pop rbp
```

```
ret
.LC0:
.string "resultat d= %d"
main:
push rbp
mov rbp, rsp
mov esi, 3
mov edi, 7
call diviser(int, int)
mov esi, eax
mov edi, OFFSET FLAT:.LC0
mov eax, 0
call printf
mov eax, 0
pop rbp
ret
```

Exercise 7 Solution

Iterative Version

```
global _start
section .text
_start:
   mov rax, [n]      ; Load n into rax
   mov rbx, 1        ; Initialize F1
   mov rdx, 0        ; Initialize F0
   call fib          ; Compute the nth Fibonacci number
   mov ebx, rbx
   mov eax, 1        ; System call number for exit
   int 0x80          ; Exit with the result in ebx
fib:
   mov rcx, 2        ; Initialize loop counter (i = 2)
loop:
   cmp rcx, rax      ; Compare i with n
   jge done          ; If i >= n, jump to done
   mov r10, rbx      ; Save Fi-1 in r10
   add rbx, rdx      ; Compute Fi = Fi-1 + Fi-2
   mov rdx, r10      ; Update Fi-2 to Fi-1
   inc rcx           ; Increment loop counter
   jmp loop          ; Repeat the loop
done:
   ret               ; Return to _start
section .data
n dq 5               ; Define n = 5
```

Recursive Version

```
section .text
global _start
_start:
   push 7            ; Push the argument (n = 7) onto the stack
   call fib
   mov ebx, eax
   mov eax, 1        ; System call number for exit
   int 0x80          ; Make the system call
fib:
   push rbp          ; Save the base pointer
   mov rbp, rsp      ; Set up the base pointer
   mov rax, [rbp + 16] ; Load the argument (n) from the stack
   cmp rax, 1        ; Compare n with 1
   jbe base_case     ; If n <= 1, jump to the base case
   ; Recursive case: fib(n) = fib(n-1) + fib(n-2)
   dec rax           ; Compute n-1
   push rax          ; Push n-1 as the argument for the first recursive call
   call fib          ; Call fib(n-1)
   mov rbx, rax      ; Save the result of fib(n-1) in rbx
   mov rax, [rbp + 16] ; Reload n
   sub rax, 2        ; Compute n-2
   push rax          ; Push n-2 as the argument for the second recursive call
   call fib          ; Call fib(n-2)
   add rax, rbx      ; Add the results of fib(n-1) and fib(n-2)
```

```
    jmp cleanup       ; Jump to cleanup
base_case:
    mov rax, [rbp + 16] ; Return n directly (f0 = 0, f1 = 1)
cleanup:
    pop rbp           ; Restore the base pointer
    ret               ; Return to the caller
```

Exercise 8 Solution

```
global _start
Section .data
v dq 5,7,1,2,4,9,7
section .text
_start: push 7
push v
call sum
xchg eax, ebx
mov         eax, 1
int         0x80
sum:
            push rbp
            mov rbp,rsp
            mov rbx,[rbp+16] ; V
            mov rcx,[rbp+24] ; 7
            dec rcx ; 6
            mov rsi,0
            mov rax,0
loopj:      add rax,[rbx+rsi]
            add rsi,8
            dec rcx
            cmp rcx,0
            jge loopj
            pop rbp
            ret 16
```

Exercise 9 Solution

```
global _start
section .text
_start: push A
call SumMat
xchg eax, ebx
```

```
mov         eax, 1
int         0x80
SumMat:
            push rbp
            mov rbp, rsp
            mov rcx, [rbp+16]     ; cx=@A
            mov rax, 0
            mov rdx, 0
            mov rsi, 0  ; si: i=0
nr:         cmp rsi, [D1]         ; if(i==D1) then quit
            je q
            mov rdi, 0 ; di: j=0
nc:         cmp rdi, [D2]         ; if(j==D2) then go to next column
            je ec
            add rax, [rcx+rdx]
            inc rdi       ; next column
            add rdx, 8
            jmp nc
ec:         inc rsi               ; next line
            jmp nr
q:          pop rbp
            ret 8
section .data
D1: dq 3
D2: dq 4
A: dq 1, 2, 2, 4
   dq 0, 1, 2, 1
   dq 1, 3, 4, 2
```

Mock tests and exams

Test 1

Exercise 01: (3pts)

A **1.2** GHz processor executes a benchmark program that uses a mixture of the following instruction types:

Instruction Type	Number of instructions	Number of Cycles/Instruction
Integer operation	300000	4
Memory access	20000	8
Floating-point	10000	10
Control (branches)	5000	5

1- How many cycles will this program take to execute? **(1 pt)**

2- How long will it take? (Execution time) **(1 pt)**

3- Calculate the CPI (cycle per instruction) of this program **(1 pt)**

Exercise 02: (3pts)

Consider a simplified 5-stage pipeline processor with the following stages:

1.ADD R3, R5, R1
2.DIV R4, R3, R2
3.MUL R3, R4, R6
4.DIV R5, R3, R1
5.SUB R2, R5, R4

Identify the data hazards in the instruction sequence (WAW, WAR, RAW). **(2pts)**

WAW	WAR	RAW

Draw the pipeline after solving its hazards. **(1pt)**

Exercise 03: (3pts)

A software application takes 60 seconds to run on a single-core processor. You determine that 70% of the program can be parallelized, while the remaining 30% must run sequentially. You are planning to use a multi-core processor to improve the execution time. If you decide to use 8 cores processor:

1. Calculate the maximum speed-up.

2. What is the expected execution time of the program in this new multicore processor?

Exercise 04: (3pts)

A rendering application takes 40 seconds to complete on computer X, which has a clock rate of 2 GHz. Computer Y is being designed to run the same application in 25 seconds, but it will require 1.5 times as many clock cycles as computer X.

1. How many clock cycles does the application take to run on computer X? **(1pts)**
2. Calculate the number of clock cycles required for computer Y to run the same application. **(1pts)**
3. What clock rate should you advise the designer to target for computer Y to achieve the desired execution time? **(1pts)**

Exercise 05 (4,5pts)

A digital computer has a memory unit with 64 bits per word. The instruction set consists of 200 different operations. Each instruction has an opcode, a memory address field, and two register address fields (for source and destination registers). This system includes 16 general-purpose registers.

1. How many bits are needed for the opcode? **(1pts)**
2. How many bits are needed to specify each register? **(1pts)**
3. How many bits are left for the memory address part of the instruction? **(1pts)**
4. What is the maximum allowable size for memory? **(1,5pts)**

Exercise 06: (2pts)

A processor operates at a clock speed of 2 GHz. The pipeline has 5 stages. What is the time required to execute 2000 instructions:

1. If the processor uses the pipeline? **(1pts)**
2. Without using the pipeline? **(1pts)**

Exercise 07: (1,5pts)

Analyze the following instructions and determine the addressing mode for each:

a) MOV AX, [1234H]:
b) MOV BX, 5000H:
c) MOV CX, [BX]:

Test 2

Exercise 1: (1,5pts)

Resume the key characteristics of Apple M Serie of microprocessors

Exercise 2 : (3pts)

Consider a simplified 5-stage pipeline processor with the following stages:

1.ADD R1, R4, R5

2.DIV R2, R1, R3

3.MUL R1, R2, R6

4.DIV R4, R1, R5

5.SUB R3, R4, R2

Identify the data hazards in the instruction sequence (WAW, WAR, RAW). **(2pts)**

WAW	WAR	RAW

Draw the pipeline after solving its hazards. **(1pts)**

Exercise 3: (3pts)

A program takes 45 seconds to execute on a single-core processor. Analysis shows that 60% of the program can be parallelized, while the remaining 40% is inherently sequential. You are planning to use a multi-core processor to improve the execution time. If you decide to use 6 core processor:

1. What is the **maximum** speed-up?
2. What is the expected execution time?

Exercise 4: (3 pts)

A **4** GHz processor executes a benchmark program that uses a mixture of the following instruction types:

Instruction Type	Number of instructions	Number of Cycles/Instruction
Integer operation	1000000	2
Memory access	100000	5
Floating-point	80000	6
Control (branches)	30000	3

1- How many cycles will this program take to execute? **(1 pt)**

2- How long will it take? (Execution time) **(1 pt)**

3- Calculate the CPI (cycle per instruction) of this program **(1 pt)**

Exercise 5: (4,5pts)

A computer system has a memory unit with 48 bits per word. The instruction set consists of 90 different operations. Instructions have an opcode, a memory address field, and a single register address field. There are 32 general-purpose registers.

1. How many bits are needed for the opcode? **(1pt)**
2. How many bits are needed to specify each register? **(1pt)**
3. How many bits are left for the memory address part of the instruction? **(1pt)**
4. What is the maximum allowable size for memory? **(1,5pts)**

Exercise 6: (3pts)

A computational task takes 10 seconds to complete on computer X, which has a clock rate of 4 GHz. Computer Y is being designed to run the same task in 6 seconds, but it will require 1.4 times as many clock cycles as computer X.

1. How many clock cycles does the application take to run on computer X? **(1pt)**
2. Calculate the number of clock cycles required for computer Y to run the same application. **(1pt)**
3. What clock rate should you advise the designer to target for computer Y to achieve the desired execution time? **(1pt)**

Exercise 7: (2pts)

A processor operates at a clock speed of 800 MHz. The pipeline has 8 stages. What is the time required to execute 5000 instructions:

1. If the processor uses the pipeline? **(1pt)**
2. Without using the pipeline? **(1pt)**

Test 3

Exercise 01: (3pts)

Consider a simplified 5-stage pipeline processor with the following stages:

1.ADD R4, R2, R6

2.DIV R1, R4, R5

3.MUL R4, R1, R3

4.DIV R2, R4, R6

5.SUB R5, R2, R1

Identify the data hazards in the instruction sequence (WAW, WAR, RAW). **(2pts)**

WAW	WAR	RAW

Draw the pipeline after solving its hazards. **(1pt)**

Exercise 02: (1,5 pts)

Write the following sequence in accumulator model and stack model: add a, c, d

Accumulator memory access model	**Stack memory access model**

Exercise 03: (3pts)

A computational task takes 100 seconds to complete on a single-core processor. It is determined that 90% of the task can be parallelized, while 10% must remain sequential. You are planning to use a multi-core processor to improve the execution time. If you decide to use 16 cores processor:

1. Calculate the maximum speed-up.
2. What is the expected execution time?

Exercise 04: (2 pts)

A processor operates at a clock speed of 1.6 GHz. The pipeline has 4 stages. What is the time required to execute 10000 instructions:

1. If the processor uses the pipeline? **(1pts)**
2. Without using the pipeline? **(1pts)**

Exercise 05: (3pts)

A software application runs in 25 seconds on computer X, which has a clock rate of 3.2 GHz. Computer Y is being designed to run the same application in 15 seconds, but it will require 1.6 times as many clock cycles as computer X.

1. How many clock cycles does the application take to run on computer X? **(1pt)**
2. Calculate the number of clock cycles required for computer Y to run the same application. **(1pt)**
3. What clock rate should you advise the designer to target for computer Y to achieve the desired execution time? **(1pt)**

Exercise 06: (4,5pts)

A processor uses 32-bit instructions. The instruction set contains 150 different operations. Each instruction includes an opcode, two register address fields (source and destination), and an immediate value field that can hold a signed integer.

1. How many bits are needed for the opcode? **(1pt)**
2. How many bits are needed to specify each register? **(1pt)**
3. How many bits are left for the memory address part of the instruction? **(1pt)**
4. What is the maximum allowable size for memory? **(1,5pts)**

Exercise 07: (3pts)

A **2.8** GHz processor executes a benchmark program that uses a mixture of the following instruction types:

Instruction Type	Number of instructions	Number of Cycles/Instruction
Integer operation	600000	3
Memory access	70000	6
Floating-point	50000	7
Control (branches)	25000	4

1- How many cycles will this program take to execute? **(1 pt)**

2- How long will it take? (Execution time) **(1 pt)**

3- Calculate the CPI (cycle per instruction) of this program **(1 pt)**

Test 4

Exercise 01: (2 pts)

A processor operates at a clock speed of 3.2 GHz. The pipeline has 7 stages. What is the time required to execute 1500 instructions:

1. If the processor uses the pipeline? **(1pt)**
2. Without using the pipeline? **(1pt)**

Exercise 02: (1,5pts)

Analyze the following instructions and determine the addressing mode for each:

a) MOV CX, [BX + SI]:.....

b) MOV BX, [4000H]: .....

c) MOV DX, [BP]:.....

Exercise 03: (4,5pts)

A computer system has a memory unit with 128 bits per word. The instruction set consists of 500 different operations. Each instruction includes an opcode, three register address fields (two source registers and one destination register), and a memory address field. There are 64 general-purpose registers.

1. How many bits are needed for the opcode? **(1pt)**
2. How many bits are needed to specify each register? **(1pt)**
3. How many bits are left for the memory address part of the instruction? **(1pt)**
4. What is the maximum allowable size for memory? **(1,5pts)**

Exercise 04: (3pts)

Consider the following instruction sequence for a simplified 5-stage pipeline processor:

1. **MUL R2, R1, R3**
2. **DIV R4, R2, R5**
3. **ADD R6, R4, R7**
4. **SUB R8, R6, R9**
5. **DIV R1, R8, R10**

Identify the data hazards in the instruction sequence (WAW, WAR, RAW). **(2pts)**

WAW	WAR	RAW

Draw the pipeline after solving its hazards. **(1pt)**

Exercise 05: (3pts)

A program takes 30 seconds to execute on computer X, which has a clock rate of 1.8 GHz. Computer Y is being designed to run the same program in 20 seconds, but it will require 2 times as many clock cycles as computer X.

1. How many clock cycles does the application take to run on computer X? (1pt)
2. Calculate the number of clock cycles required for computer Y to run the same application. (1pts)
3. What clock rate should you advise the designer to target for computer Y to achieve the desired execution time? (1pts)

Exercise 06 : (3pts)

A **1.5** GHz processor executes a benchmark program that uses a mixture of the following instruction types:

Instruction Type	Number of instructions	Number of Cycles/Instruction
Integer operation	800000	4
Memory access	30000	9
Floating-point	20000	8
Control (branches)	15000	5

1- How many cycles will this program take to execute? **(1 pt)**

2- How long will it take? (Execution time) **(1 pt)**

3- Calculate the CPI (cycle per instruction) of this program **(1 pt)**

Exercise 07: (3pts)

A simulation program takes 80 seconds to run on a single-core processor. You find that 50% of the program can be parallelized, while the remaining 50% is sequential. You are planning to use a multi-core processor to improve the execution time. If you decide to use 4 core processor:

1. What is the maximum speed-up?
2. What is the expected execution time?

Test 5

Exercise 01: (3pts)

Consider the following instruction sequence for a simplified 5-stage pipeline processor:

1. LOAD R1, 0(R2)
2. ADD R3, R1, R4
3. STORE R3, 0(R5)
4. MUL R4, R3, R6
5. SUB R7, R4, R8

Identify the data hazards in the instruction sequence (WAW, WAR, RAW). **(2pts)**

WAW	WAR	RAW

Draw the pipeline after solving its hazards. **(1pts)**

Exercise 02: (3 pts)

A 3 GHz processor executes a benchmark program that uses a mixture of the following instruction types:

Instruction Type	Number of instructions	Number of Cycles/Instruction
Integer operation	500000	2
Memory access	40000	7
Floating-point	60000	5
Control (branches)	10000	3

1- How many cycles will this program take to execute? **(1 pt)**

2- How long will it take? (Execution time) **(1 pt)**

3- Calculate the CPI (cycle per instruction) of this program **(1 pt)**

Exercise 03: (3pts)

Your favorite application runs in 20 seconds on computer X, which has a clock rate of 2.5 GHz. You want to assist a computer designer in building computer Y, which should run the same application in 12 seconds. The designer has found that due to architectural changes, computer Y will require 1.8 times as many clock cycles to execute the application as

computer X.

1. How many clock cycles does the application take to run on computer X? **(1pt)**
2. Calculate the number of clock cycles required for computer Y to run the same application. **(1pt)**
3. What clock rate should you advise the designer to target for computer Y to achieve the desired execution time? **(1pt)**

Exercise 04: (3pts)

A data processing application takes 120 seconds to execute on a single-core processor. It is estimated that 85% of the program can be parallelized, while 15% must run sequentially. You are planning to use a multi-core processor to improve the execution time. If you decide to use 8 core processor:

1. Calculate the maximum speed-up?
2. What is the expected execution time?

Exercise 05: (2 pts)

A processor operates at a clock speed of 1.2 GHz. The pipeline has 6 stages. What is the time required to execute 2500 instructions:

1. If the processor uses the pipeline? **(1pts)**
2. Without using the pipeline? **(1pts)**

Exercise 06: (1,5 pts)

Resume the key characteristics of Apple A Serie of microprocessors

Exercise 07: (4,5pts)

A microprocessor uses 64-bit instructions. The instruction set consists of 75 different operations. Each instruction includes an opcode, a memory address field, and a single register address field. There are 16 general-purpose registers.

1. How many bits are needed for the opcode? **(1pts)**
2. How many bits are needed to specify each register? **(1pts)**
3. How many bits are left for the memory address part of the instruction? **(1pts)**
4. What is the maximum allowable size for memory? **(1,5pts)**

Test 6

Exercise 01: (3pts)

A rendering task takes 200 seconds to complete on a single-core processor. Analysis reveals that 75% of the task can be parallelized, while 25% is inherently sequential. You are planning to use a multi-core processor to improve the execution time. If you decide to use 12 core processor:

1. What is the maximum speed-up?
2. What is the expected execution time?

Exercise 02: (3pts)

A data processing program runs in 18 seconds on computer X, which has a clock rate of 2.4 GHz. Computer Y is being designed to run the same program in 12 seconds, but it will require 1.7 times as many clock cycles as computer X.

1. How many clock cycles does the application take to run on computer X? **(1pt)**
2. Calculate the number of clock cycles required for computer Y to run the same application. **(1pt)**
3. What clock rate should you advise the designer to target for computer Y to achieve the desired execution time? **(1pt)**

Exercise 03: (3pts)

Consider the following instruction sequence for a simplified 5-stage pipeline processor:

1. ADD R3, R1, R2
2. SUB R4, R3, R5
3. MUL R3, R4, R6
4. DIV R5, R3, R7
5. ADD R6, R5, R8

Identify the data hazards in the instruction sequence (WAW, WAR, RAW). **(2pts)**

WAW	WAR	RAW

Draw the pipeline after solving its hazards. **(1pts)**

Exercise 04: (2 pts)

A processor operates at a clock speed of 500 MHz. The pipeline has 10 stages. What is the time required to execute 800 instructions:

1. If the processor uses the pipeline? **(1pts)**
2. Without using the pipeline? **(1pts)**

Exercise 05: (1,5pts)

Write the following sequence in accumulator model and stack model: add a, c, d

Register- memory access model	**Load-Store memory access model**

Exercise 06: (3pts)

A **2** GHz processor executes a benchmark program that uses a mixture of the following instruction types:

Instruction Type	**Number of instructions**	**Number of Cycles/Instruction**
Integer operation	400000	3
Memory access	50000	8
Floating-point	30000	6
Control (branches)	20000	4

1- How many cycles will this program take to execute? **(1pt)**

2- How long will it take? (Execution time) **(1 pt)**

3- Calculate the CPI (cycle per instruction) of this program **(1 pt)**

Exercise 07: (4,5pts)

A computer system has a memory unit with 256 bits per word. The instruction set consists of 1000 different operations. Each instruction includes an opcode, four register address fields (three source registers and one destination register), and a memory address field. There are 128 general-purpose registers.

1. How many bits are needed for the opcode? **(1pt)**
2. How many bits are needed to specify each register? **(1pt)**
3. How many bits are left for the memory address part of the instruction? **(1pt)**
4. What is the maximum allowable size for memory? **(1,5pt)**

Test 7

Exercise 1: (4.5pts)

1. Correct the errors in the following program:

```
global _start
section .text
_start
  mov rsi; array
  mov rcx, 5
outer_loop
  mov rdx, rcx
  dec rdx
inner_loop:
  mov rax, [rsi]
  mov rbx, [rsi+8]
  cmp rax, rbx
  jle no_swap_
  xchg rax rbx
  mov [rsi], rax
  mov [rsi+8]; rbx
no_swap
  add rsi, 8
  dec rdx
  jnz inner_loop
  sub rsi, 5
  dec rcx
  jnz outer_loop
exit:
  mov eax, 60
  xor edi, edi
  syscall
section.data
array dq 5, 3, 8, 1, 4
```

2. What does the program do?

Exercise 2 : (4pts)

Write a NASM program that calculates the sum of the minimum and maximum values of an array T[6] and stores the result in R9. Additionally, stores the minimum value in R10 and the maximum value in R11.

Assume the array T contains the following values: T dq 5, 2, 8, 1, 7, 3

Exercise 03 : (2 pts)

A processor has a 32-bit memory address space. The memory is broken into blocks of 64 bytes each. The computer also has a cache capable of storing 8K bytes.

1. How many blocks can the cache store?
2. Assuming the cache uses direct mapping, how many bits are there in each of the TAG, BLOCK, and BYTE OFFSET fields of the address?

Exercise 04 : (9,5 pts)

A computer system uses 16-bit memory addresses. It has a 2K-byte cache organized in a direct-mapped manner. Compare two configurations:

- **Configuration A**: 32 bytes per cache block.
- **Configuration B:** 64 bytes per cache block.

1. For each configuration, calculate the number of bits in each of the Tag, Block, and Word fields of the memory address. (2,5pts)

When a program is executed, the processor reads data sequentially from the following word addresses (in decimal): 128, 256, 4096, 4160, 128, 4096.

2. Assume that the cache is initially empty. For each configuration, indicate whether the cache access will result in a hit or a miss for each address. (6pts)
3. Compare the hit ratio and miss ratio for the two configurations. (1pt)

Test 8

Exercise 1 : (6.5pts)

1. Correct the errors in the following program:

```
global _start
section text
_start_
  mov rsi array;
  mov rcx, 5
  xor rax, rax
sum_loop:
  add rax [rsi]
  add rsi 4
  loop sum_loop
  mov rbx; rax
  mov eax; 60
  xor edi, edi
  syscall_
section.data
array dq 1, 2, 3, 4, 5
```

2. What does the program do?

Exercise 2 : (3pts)

Write a NASM program that calculates the average of the elements in an array T[6] and stores the result in R9. Additionally, stores the sum of all elements in R10 and the number of elements in R11. Assume the array T contains the following values: T dq 4, 6, 8, 10, 12, 14

Exercise 03 : (1pts)

A processor has a 32-bit memory address space. The memory is broken into blocks of 128 bytes each. The computer also has a cache capable of storing 32K bytes.

1. How many blocks can the cache store?
2. Assuming the cache uses 2-way set-associative mapping, how many bits are there in each of the TAG, SET, and BYTE OFFSET fields of the address?

Exercise 04 : (9,5 pts)

A computer system uses 18-bit memory addresses. It has a 4K-byte cache organized in a direct-mapped manner. Compare two configurations:

- **Configuration A**: 32 bytes per cache block.
- **Configuration B:** 64 bytes per cache block.

1. For each configuration, calculate the number of bits in each of the Tag, Block, and Word fields of the memory address. (2,5pts)

When a program is executed, the processor reads data sequentially from the following word addresses (in decimal): 0, 1024, 2048, 1024, 4096, 2048

2. Assume that the cache is initially empty. For each configuration, indicate whether the cache access will result in a hit or a miss for each address. (6pts)
3. Compare the hit ratio and miss ratio for the two configurations. **(1pt)**

Test 9

Exercise 1 : (4.5pts)

1. Correct the errors in the following program:

```
global _start
section.text
_start:
  mov rax; n
  mov rbx 1
  mov rdx 0
  call t1
  xchg eax, ebx
  mov eax, 1
  int 0*80
;t1:
  mov rcx, 2
t2
  add r10, rdx
  add r10, rbx
  mov rdx, rbx,
  mov rbx, r10;
  inc rcx
  cmp rcx; rax
  jl t2 :
  ret 8
section data
n dq 5
```

2. What does the program do?

Exercise 2 : (4pts)

Write a NASM program that counts the number of odd numbers in an array T[6] and stores the result in R9. Additionally, stores the smallest odd number in R10 and the largest odd number in R11. Assume the array T contains the following values: T dq 3, 4, 7, 8, 9, 10

Exercise 03 : (2pts)

A processor has a 32-bit memory address space. The memory is broken into blocks of 128 bytes each. The computer also has a cache capable of storing 16K bytes.

1. How many blocks can the cache store?
2. Assuming the cache uses direct mapping, how many bits are there in each of the TAG, BLOCK, and BYTE OFFSET fields of the address?

Exercise 04 : (9,5 pts)

A computer system uses 20-bit memory addresses. It has a 16K-byte cache organized in a direct-mapped manner. Compare two configurations:

- **Configuration A**: 64 bytes per cache block.
- **Configuration B:** 128 bytes per cache block.

1. For each configuration, calculate the number of bits in each of the Tag, Block, and Word fields of the memory address. **(2,5pts)**

When a program is executed, the processor reads data sequentially from the following word addresses (in decimal): 0, 64, 256, 4160, 64, 4160, 256

2. Assume that the cache is initially empty. For each configuration, indicate whether the cache access will result in a hit or a miss for each address. (6pts)
3. Compare the hit ratio and miss ratio for the two configurations. (1pt)

Test 10

Exercise 1 : (4.5pts)

1. Correct the errors in the following program:

```
    global _start
    section text
    start_:
      mov rax; n
      call i_p
      mov rax, 60
      mov rdi, rax
      syscall
    i_p:
      cmp rax, 2
      jl n_p
      mov rbx 2
    check_d:
      mov rdx, 0
      div rbx,
      cmp rdx, 0
      je n_p:
      inc rbx
      cmp rbx, rax
      jl check_d
      mov rax, 1
      ret
    n_p:
      xor rax, rax
      ret
    section.data
    n dq, 17
```

2. What does the program do?

Exercise 2 : (4 pts)

Write a NASM program that finds the second-largest value in an array T[6] and stores the result in R9. Additionally, stores the largest value in R10, the smallest value in R11. Assume the array T contains the following values: T dq 5, 1, 9, 3, 7, 2

Exercise 03 : (5pts)

A processor has a 32-bit memory address space. The memory is broken into blocks of 64 bytes each. The computer has a two-level cache hierarchy:

- **L1 Cache**: 16K bytes, direct-mapped.
- **L2 Cache**: 64K bytes, 4-way set-associative.

For each level of the cache hierarchy:

1. How many blocks can the cache store?
2. How many bits are there in each of the TAG, BLOCK/SET, and BYTE OFFSET fields of the address?

Exercise 04 : (5,5pts)

A computer system uses 16-bit memory addresses. It has a 1K-byte cache organized in a 2-way set-associative manner with 16 bytes per cache block. Assume that the size of each memory word is 1 byte.

1. Calculate the number of bits in each of the Tag, Set, and Word fields of the memory address.

When a program is executed, the processor reads data sequentially from the following word addresses (in decimal):32, 64, 512, 528, 32, 512.

2. Assume that the cache is initially empty. For each of the above addresses, indicate whether the cache access will result in a hit or a miss.

Test 11

Exercise 1 : (4.5pts)

1. Correct the errors in the following program:

```
global _start
section text
_start:
  mov rsi; string
  xor rcx rcx
strlen_loop
  cmp byte [rsi], 0
  je end_strlen
  inc rcx,
  inc rsi
  jmp strlen_loop
endstrlen_:
  mov rbx, rax
  mov eax 60
  xor edi edi
  syscall
section.data
string db "Hello", 0
```

2. What does the program do?

Exercise 2 : (4pts)

Write a NASM program that reverses the elements of an array T[6] and stores the reversed array back into T. Additionally, stores the original first element (before reversal) in R10 and the original last element (before reversal) in R11. Assume the array T contains the following values: T dq 1, 2, 3, 4, 5, 6

Exercise 03 : (5pts)

A processor has a 32-bit memory address space . The computer has a cache capable of storing 32K bytes . Compare two configurations:

- **Configuration A** : Blocks of 128 bytes each.
- **Configuration B** : Blocks of 256 bytes each.

For each configuration:

1. How many blocks can the cache store?
2. Assuming the cache uses direct mapping, how many bits are there in each of the TAG, BLOCK, and BYTE OFFSET fields of the address?

Exercise 04 : (5,5pts)

A computer system uses 16-bit memory addresses . It has a 4K-byte cache organized in a direct-mapped manner with 32 bytes per cache block . Assume that the size of each memory word is 1 byte .

1. Calculate the number of bits in each of the Tag , Block , and Word fields of the memory address.

When a program is executed, the processor reads data sequentially from the following word addresses (in decimal): 64, 96, 4096, 4112, 64, 4096.Assume that the cache is initially empty.

2. For each of the above addresses, indicate whether the cache access will result in a hit or a miss.

Test 12

Exercise 1: (4.5pts)

1. Correct the errors in the following program:

```
global _start
section .text
_start:
  mov rax; n
  mov rbx. 0
  mov rcx 1
  call f1
  mov eax 60
  xor edi edi
  syscall
f1:
  cmp rax, 1
  jle f2
  dec rax
  push rbx
  push rcx
  call f1
  pop rcx
  pop rbx
  imul rbx rcx
f2
  ret
section data
n dq 7
```

2. What does the program do?

Exercise 2 : (4pts)

Write a NASM program that calculates the average of all even numbers in an array T[6] and stores the result in R9. Additionally, stores the count of even numbers in R10 and the sum of even numbers in R11. Assume the array T contains the following values: T dq 1, 4, 7, 8, 10, 15

Exercise 03 : (3,5pts)

A processor has a 32-bit memory address space. The memory is broken into blocks of 32 bytes each. The computer has a cache capable of storing 64K bytes.

1. How many blocks can the cache store?
2. Assuming the cache uses 8-way set-associative mapping, how many bits are there in each of the TAG, SET, and BYTE OFFSET fields of the address?

Exercise 04 : (8pts)

A computer system uses 16-bit memory addresses. It has a 4K-byte cache organized in a 2-way set-associative manner with 64 bytes per cache block. Assume that the size of each memory word is 1 byte.

1. Calculate the number of bits in each of the Tag, Set, and Word fields of the memory address.

When a program is executed, the processor reads data sequentially from the following word addresses (in decimal): 128, 256, 4096, 4160, 128, 4096.

2. Assume that the cache is initially empty. For each of the above addresses, indicate whether the cache access will result in a hit or a miss.

Test 13

Put (X) next to the appropriate answer(s).

1- Which of the following best describes cache memory? (1pt)		**2- Quantum computers use which basic unit of information? (1pt)**	
A. A type of secondary storage used for long-term data retention		**A.** Bit	
B. High-speed memory that serves as a buffer between RAM and the CPU		**B.** Qubit	
C. Memory that stores data permanently even after power is turned off		**C.** Byte	
D. A component used only in GPUs for fast rendering		**D.** Nibble	
3- Which statement best describes CISC architecture? (1pt)		**4- In pipelining, what does "hazard" refer to? (1pt)**	
A. Uses simple instructions that can be completed in one clock cycle		**A.** When the pipeline must wait due to dependencies or conflicts	
B. Has a large set of complex instructions.		**B.** When the CPU runs out of instructions to process	
C. Is primarily used in modern mobile devices		**C.** When the cache becomes full	
D. Focuses on hardware complexity to reduce the number of instructions per program		**D.** When two instructions finish executing at the same time	
5- Which addressing mode uses the value of the operand directly in the instruction? (1pt)		**6- The first generation of computers used which technology? (1pt)**	
A. Direct addressing		**A.** Microprocessors	
B. Register addressing		**B.** Vacuum tubes	
C. Indirect addressing		**C.** Integrated circuits	
D. Immediate addressing		**D.** Transistors	
7- Which of the following cache mapping techniques allows a block of main memory to be placed in any cache line? (1pt)		**8- Which type of memory retains data without needing constant refreshing? (1pt)**	
A. Direct mapping		**A.** DRAM	
B. Set-associative mapping		**B.** SDRAM	

C. Fully associative mapping	**C.** SRAM
D. Random mapping	**D.** VRAM
9- Swapping is a function of:..... (1pt)	**10- Which of the following is a characteristic of an embedded system? (1pt)**
A. Cache memory management	**A.** It is designed to perform a wide range of general-purpose tasks.
B. GPU memory allocation	**B.** It is designed to perform specific functions or tasks.
C. Pipeline optimization	**C.** It requires constant user interaction for its operation
D. Virtual memory management	**D.** It is used only for high-performance computing tasks.
11- A computer system uses a 20-bit memory address. It has an 8K-byte cache organized in a direct-mapped manner with 256 bytes per block. How many bits are used for the Tag, Block Index, and Word Offset fields in this cache? (1pt)	**12- For the same computer system in question 11, If the processor accesses the following addresses sequentially: 1024, 2068, 6164, 6272, which sequences of hits and misses will occur? Assume the cache is initially empty. (1pt)**
A. Tag: 7 bits, Block Index: 8 bits, Word Offset: 5 bits	**A.** Miss, Miss, Miss, Hit
B. Tag: 7 bits, Block Index: 5 bits, Word Offset: 8 bits	**B.** Miss, Miss, Miss, Miss
C. Tag: 5 bits, Block Index: 7 bits, Word Offset: 8 bits	**C.** Miss, Hit, Miss, Miss
D. Tag: 8 bits, Block Index: 5 bits, Word Offset: 7 bits	**D.** Miss, Miss, Hit, Miss

13- Correct the following program **1,5pts**

```
global _start
section .data
a dq 3
section text
_start:
        push qword a
        call fi_
        xchg eax, ebx
        mov     eax, 1
        mov rax, [rbp+16]
        cmp rax 1
        jg fo_
        mov rax, 1
        jump fin_
fo_: sub rax, 1
        push rax
        call fi_
        mov rcx, [rbp+16],
```

14- What does the following program do? (0,5pt)

A. Multiplication of two numbers	
B. Power of two numbers	
C. Factorial	
D. Fibonnaci (iterative)	

15- Where is the final result stored? (0,5pt)

A. Rax	
B. Rcx	
C. Stack	
D. [rsp]	

int 0x80 fi: push rbp mov rbp, rsp	imul rax, rcx fin_: pop rbp ret 8	**16- What is the returned result? (0,5pt)**
		A. 8
		B. 16
		C. 6
		D. 5

17- Suppose the CPU clock of a machine is 3.2 GHz and this machine has a 6-stage pipeline (without hazards). What is the CPU execution time when this machine executes 2000 instructions? What is the execution time with pipelining and without pipelining. (1pt)	
A. **Without** Pipelining: 3750ms. **With** Pipelining: 626.5625ms	
B. **Without** Pipelining: 3750ms. **With** Pipelining: 626.5625ns	
C. **Without** Pipelining: 3750ns. **With** Pipelining: 626.5625ns	
D. **Without** Pipelining: 3750µs. **With** Pipelining: 626.5625µs	

18- You are evaluating the performance improvement of upgrading the GPU in a gaming computer. The new GPU is 10 times faster than the old one. However, 25% of the tasks are CPU-bound. What is the overall speedup obtained using the improved GPU? (1pt)		**19- Suppose the parallelizable portion of the program is 90%. How many processors are needed to achieve a speedup of 5x? (1pt)**	
A. 3,08		**A.** 8	
B. 1,29		**B.** 9	
C. 3,80		**C.** 7	
D. 1,92		**D.** 6	
20- Consider the cache has 7 blocks. For the memory references 20, 35, 40, 50, 25, 30, 35, 40, 45, 50, 55, 35, 60, 65, 70, 80. What is the hit ratio for the LRU cache replacement algorithms (1pt)		**21- Which of the following memory blocks will not be in the cache at the end of the sequence? Given in question 20. (1pt)**	
A. **Hit** rate: 25%, **miss** rate: 75%		**A.** 35	
B. **Hit** rate: 35%, **miss** rate: 65%		**B.** 40	
C. **Hit** rate: 45%, **miss** rate: 55%		**C.** 50	
D. **Hit** rate: 75%, **miss** rate: 25%		**D.** 55	

Revision Quiz

1. Which of the following best describes the Von Neumann bottleneck?

A) Limited parallelism due to multiple cores sharing the same memory

B) Delay due to instruction-level parallelism

C) Limited throughput due to a single shared bus for instructions and data

D) Overhead from frequent cache misses

Answer: C

2. What significant change did the transition from 32-bit to 64-bit processors introduce?

A) Doubling of instruction execution speed

B) Increased support for SIMD operations

C) Larger addressable memory space

D) Hardware support for virtual memory

Answer: C

3. In the Intel 8086 architecture, what is the size of a segment?

A) 64 KB

B) 1 MB

C) 16 KB

D) 1 KB

Answer: A

4. In pipelining, what is a structural hazard?

A) Occurs when data is not ready

B) Occurs when multiple instructions require the same hardware resource

C) Happens due to incorrect branch prediction

D) Caused by compiler optimization

Answer: B

5. Which of the following is not an addressing mode in x86 architecture?

A) Immediate

B) Indirect

C) Displacement

D) Pipeline

Answer: D

6. Which cache mapping technique is most efficient for reducing conflict misses?

A) Direct mapped cache

B) Fully associative cache

C) Set-associative cache

D) Write-through cache

Answer: B

7. What characteristic distinguishes an embedded system from a general-purpose computer?

A) Use of RISC processors

B) Lack of storage

C) Task-specific design and limited resources

D) Use of quantum computing principles

Answer: C

9. In multicore systems, what is the main challenge with shared caches?

A) Reduced memory bandwidth

B) Increased power consumption

C) Maintaining cache coherence

D) Instruction set compatibility

Answer: C

10. Which of the following best describes a quantum bit (qubit)?

A) A classical bit stored in cache memory

B) A bit that can only be in 0 or 1

C) A quantum state that can be both 0 and 1 simultaneously

D) A type of error-corrected transistor

Answer: C

11. What generation of computers introduced integrated circuits (ICs)?

A) First

B) Second

C) Third

D) Fourth

Answer: C

12. What is Moore's Law primarily concerned with?

A) Cost of transistors

B) Size of memory

C) Number of transistors on a chip

D) Speed of processors

Answer: C

13. The Intel 4004 microprocessor had how many bits?

A) 4-bit

B) 8-bit

C) 16-bit

D) 32-bit

Answer: A

14. What technique is used to minimize control hazards?

A) Dynamic scheduling

B) Branch prediction

C) Loop unrolling

D) Register renaming

Answer: B

15. What is pipeline stall?

A) A technique to increase instruction throughput

B) A delay due to hazard resolution

C) Reordering of instructions

D) Overclocking the pipeline

Answer: B

16. A GPU can be considered a:

A) General-purpose core

B) Co-processor optimized for serial tasks

C) Specialized parallel co-processor

D) Cache controller

Answer: C

17. Heterogeneous multicore architecture involves:

A) Cores with identical performance

B) Using only ARM cores

C) A mix of high-power and low-power cores

D) Disabling cache coherence

Answer: C

18. What does "cache hit" mean?

A) Data is not found in cache

B) Data is found in cache

C) Cache overflow occurs

D) Cache line is replaced

Answer: B

19. Virtual memory provides:

A) Larger physical RAM

B) More virtual CPU cores

C) Illusion of a large contiguous memory

D) Faster memory access

Answer: C

20. Page faults occur when:

A) Cache is full

B) Data is in RAM but not in cache

C) Data is not in main memory

D) CPU stalls due to hazards

Answer: C

21. What is the key difference between RAM and ROM?

A) RAM is permanent

B) ROM is volatile

C) ROM is read-only

D) RAM stores BIOS

Answer: C

22. DRAM is slower than SRAM because:

A) It uses less power

B) It requires refreshing

C) It is more expensive

D) It stores data in flip-flops

Answer: B

23. An immediate addressing mode means:

A) Address is computed using registers

B) Operand is stored in memory

C) Operand is in the instruction itself

D) Indirection through pointers

Answer: C

24. Indexed addressing is useful for:

A) Jump instructions

B) Loop counters

C) Stack operations

D) Array access

Answer: D

25. Which of these is NOT typically found in embedded systems?

A) Operating system

B) Specialized hardware

C) General-purpose GPU

D) Flash memory

Answer: C

26. Which language is most commonly used in embedded systems?

A) Java

B) Python

C) C

D) Rust

Answer: C

27. A quantum computer's strength lies in:

A) Deterministic computing

B) Parallelism through multithreading

C) Superposition and entanglement

D) Faster clock speed

Answer: C

28. What kind of gates are used in quantum circuits?

A) Logic gates

B) NAND-only gates

C) Quantum gates like Hadamard

D) XOR only

Answer: C

29. What is "entanglement" in quantum computing?

A) A form of memory error

B) Linking multiple CPUs

C) A phenomenon where qubits affect each other

D) Merging classical and quantum algorithms

Answer: C

30. Harvard architecture separates:

A) ALU and CU

B) Instruction and data memory

C) Registers and RAM

D) Control and status registers

Answer: B

31. In memory hierarchy, which is typically the slowest?

A) L1 Cache

B) Main memory

C) Registers

D) Secondary storage

Answer: D

32. A SIMD processor performs:

A) Different operations on different data

B) The same operation on multiple data

C) Sequential execution

D) Pipeline flushing

Answer: B

33. A stack is a:

A) FIFO structure

B) LIFO structure

C) RAM replacement

D) Register in CPU

Answer: B

34. EEPROM differs from ROM because:

A) It is volatile

B) It can be reprogrammed electrically

C) It stores cache

D) It is read-only forever

Answer: B

35. Memory-mapped I/O allows:

A) ROM access during boot

B) Shared CPU and GPU memory

C) I/O devices to use address space

D) DMA transfers only

Answer: C

36. Which instruction is likely to cause a context switch?

A) NOP

B) RET

C) INT

D) JMP

Answer: C

Bibliography

1. **Structured Computer Organization**, Author: Andrew S. Tanenbaum, Edition: Pearson, 2012
2. **Computer Organization and Design: The Hardware/Software Interface**, Authors: David A. Patterson, John L. Hennessy, Edition: Morgan Kaufmann.
3. **Architecture et Technologie des Ordinateurs: Cours et Exercices Corrigés - 6e édition**, Authors: Paolo Zanella, Yves Ligier, Emmanuel Lazard, 2018
4. **Embedded Systems Design.** Author: Brock J. LaMeres, Ph.D., Springer, 479 pages.
5. **Modern Processor Design: Fundamentals of Superscalar Processors.** Authors: John Paul Shen and Mikko H. Lipast
6. Lecture: Fonctionnement et performance des processeurs. Eric Cariou; Département Informatique. Université de Pau et des Pays de l'Adour.
7. **The Intel Microprocessors: 8086/8088, 80186/80188, 80286, 80386, 80486, Pentium, Pentium Pro** Processor, Pentium II, Pentium III, Pentium 4, and Core2 with 64-bit Extensions, Author: Barry B. Brey, Edition: Pearson
8. NASM (Netwide Assembler): https://www.nasm.us/
9. **Logisim**: https://cburch.com/logisim/
10. **Arduino**: https://wokwi.com/
11. **Memory:** https://www3.ntu.edu.sg/home/smitha/ParaCache/Paracache/start.html
12. **PC Building Simulation**: https://www.pcbuildingsim.com/
13. **PC building Simulator with price estimation**: https://pcpartpicker.com/
14. **Embedded systems:** https://www.edx.org/learn/embedded-systems