Table of Contents of Chapter 2

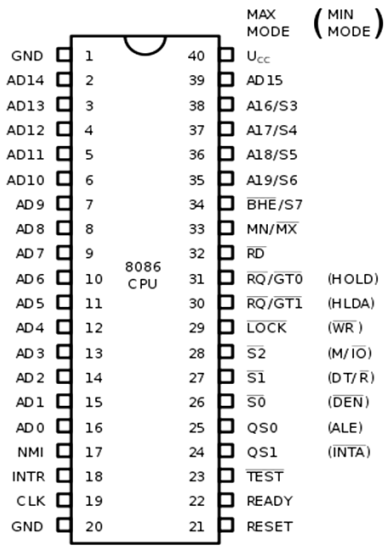
Chapter 2: Internal Architecture of the Intel 8086 Microprocessor

1 Introduction

The Intel 8086 is a 16-bit CISC microprocessor manufactured by Intel. It was launched in May 1978. It is the first processor in the x86 family, which has become the most widely used processor architecture in the world of personal computers, workstations, and server systems.
Intel's 8086 processor serves as the foundation for the Pentium processors. Subsequent (PC) processors were gradually built by adding additional instructions and features while preserving the specifications of the previous processor. It is this approach of adapting processors at each stage that allows an old program written for an 8086 to run on a new computer equipped with a Pentium 4.

2 External appearance of the 8086 microprocessor

It comes in the form of a 40-pin DIP (Dual In-line Package) package (figure 2.1).



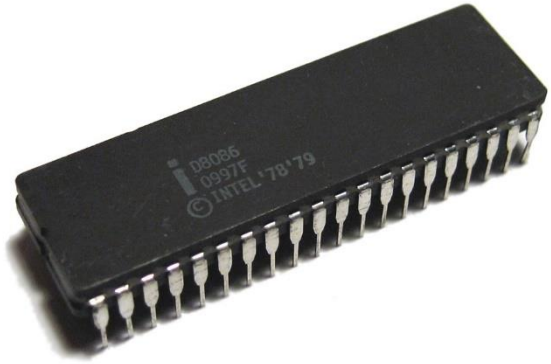


Figure 2.1. Pinout of the intel 8086 microprocessor in DIP-40 package

3 Features of the 8086

It is based on 16-bit registers, and has a 16-bit data bus and a 20-bit address bus, which allows it to address 1 MB. It contains 29,000 transistors engraved in 3 µm. Its computing power varies from 0.33 MIPS (when clocked at 4.77 MHz as in the IBM PC) up to 0.75 MIPS for the 10 MHz version. The i8086 microprocessor is characterized by:

- 16-bit structure.
- Addressing capacity of 1 MB.
- 14 internal 16-bit registers.
- 7 addressing modes.
- Operations on bits, bytes, words and character strings.
- Signed or unsigned arithmetic.
- Binary arithmetic or BCD, on 8 or 16 bits.

Definition: A register is a small part of memory integrated into the microprocessor, for the purpose of receiving information, including addresses and data stored during the execution of a program.

4 Internal architecture of the 8086

The 8086 processor has been partitioned into two separate functional units: Bus Interface Unit **(BIU)** and Execution Unit **(EU)**. The main reason for this separation is to ensure simultaneous operation of the two units, hence an increase in the processing speed of the processor.

- The BIU must interact with memory and input/output devices to retrieve instructions and data required

by the UE.

- The EU is responsible for carrying out the program instructions and the required processing.

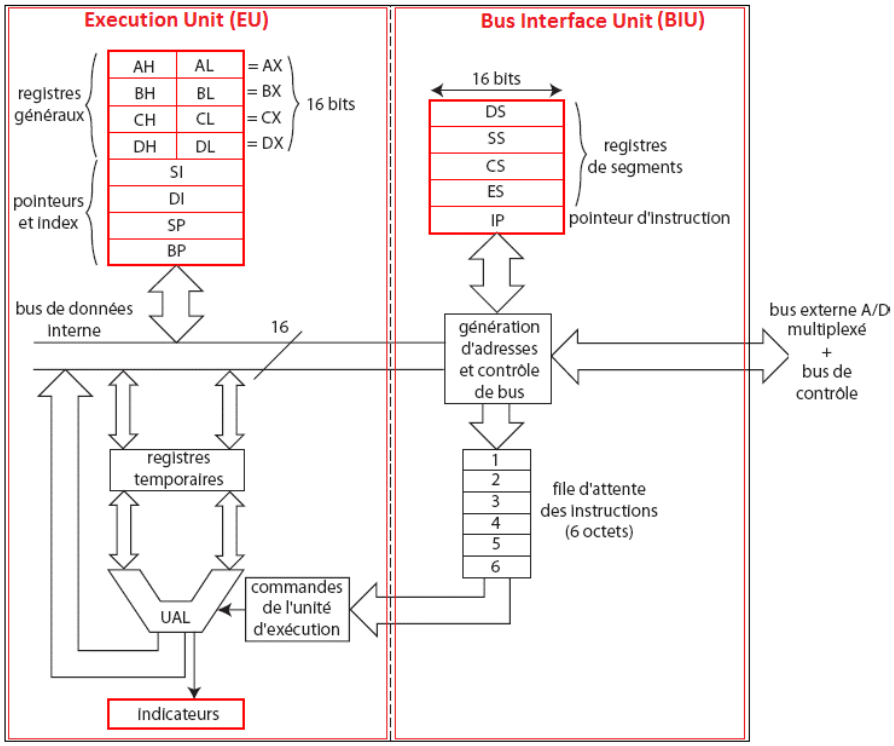


Figure 2.2. Internal architecture of the 8086.

4.1 Description of the Bus Interface Unite (BIU)

The **BIU** includes an instruction queue managed in FIFO (**F**irst **I**n **F**irst **O**ut), the segment registers (**CS, DS, SS, ES**) and the instruction pointer (**IP**). The role of the BIU is to:

- Searches for instructions to execute in memory and stores them in the FIFO queue.
- Calculate physical addresses on 20 bits.
- Carry out data transfer with memory

4.1.1 Segment Registers

The 8086 has four 16-bit segment registers: CS, DS, ES, and SS. These registers are responsible for selecting different memory segments by pointing to the beginning of each one. Each memory segment cannot exceed 65536 = 216 bytes **(64 KB)**.

a) CS Register (Code Segment): It points to the segment that contains the code instructions of the current program.

b) DS Register (Data Segment): The data segment register points to the segment of global variables in the program, and of course, its size cannot exceed 64 KB.

c) ES Register (Extra Segment): The additional data segment register ES is used by the microprocessor when accessing other registers has become difficult or impossible to modify data. This segment is also used for storing character strings.

d) SS Segment (Stack Segment): The SS register points to the stack. The stack is a memory area where registers, addresses, or data can be saved for retrieval after the execution of a subroutine or an interrupt program.

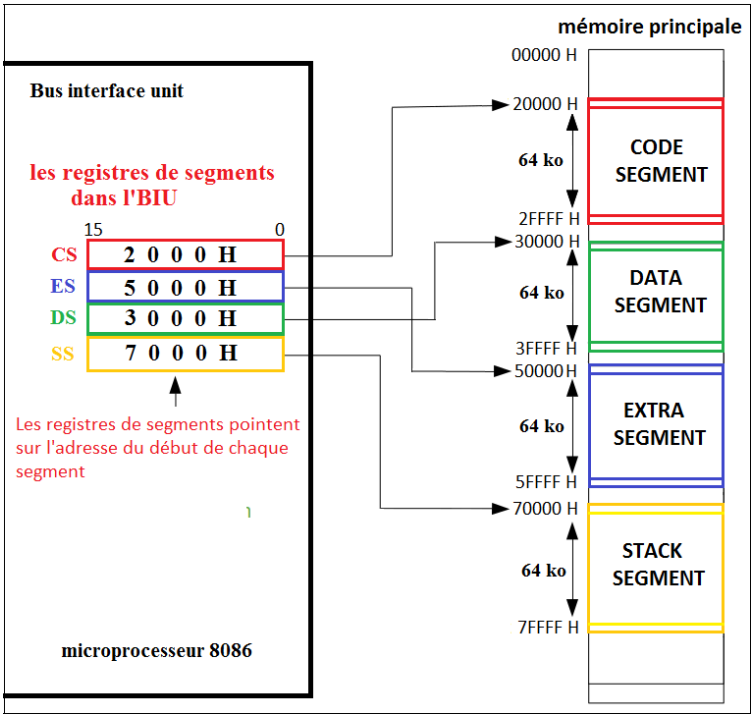


Figure 2.3. Memory Segmentation

4.1.1.1 Instruction pointer (IP) register

Also known as the **program counter**, it contains the address of the memory location where the next instruction to be executed is located. The **IP** register is constantly updated after the execution of each instruction to point to the next instruction.

4.2 Description of Execution Unit (EU)

The EU (Execution Unit) consists of the ALU (Arithmetic and Logic Unit), general registers (AX, BX, CX, DX), addressing registers (SP, BP, SI, DI), the status register, and the instruction decoder. The role of the EU is to:

- Extract instruction codes from the queue located in the BIU and execute them.
- Provide operand addresses to the UIB by specifying the relevant segment and providing the offset within that segment.

4.2.1 Arithmetic and Logic Unit (ALU)

is a complex circuit that performs logical functions (AND, OR, Comparison, Shift, etc.) as well as arithmetic functions (Addition, Subtraction).

4.2.2 General-Purpose Registers

The general registers can be used in all arithmetic and logical operations that the programmer inserts into the assembly code. A full register has a size of 16 bits. As shown in Figure 2.4, each register is actually divided into two separate 8-bit registers. This way, we can use a portion of the register if we want to store a value that does not exceed 8 bits.

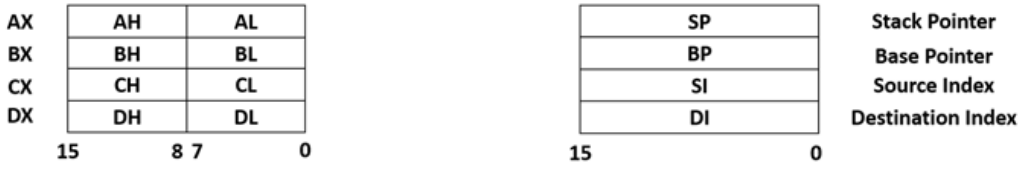


Figure 2.4. General-Purpose Registers

a) Data group :

This group consists of 4 registers of 16 bits each (**AX, BX, CX,** and **DX**). Each register can be divided into two 8-bit registers (**AH, AL, BH, BL, CH, CL, DH,** and **DL**).

AX register (Accumulator or working register): All data transfer operations with input-output, as well as string processing, are performed in this register. Arithmetic and logical operations also use this register. BCD conversions of the result of arithmetic operations (addition, subtraction, multiplication, and division) are carried out in this register.

BX register (Base register): It is used for data addressing, typically containing an offset address relative to a reference address (data segment DS). It can also be used for code conversion.

CX register (Counter): When executing a loop, a loop counter is often used to count the number of iterations, and the CX register is designed to serve as a counter during loop instructions.

DX register: It is used for multiplication and division operations, but primarily to hold the number of an input/output port for addressing I/O interfaces.

b) Pointer and index group:

These registers are specifically designed for working with elements in memory and often have increment and decrement properties.

SI (Source Index): It is used to point to memory and typically forms an offset relative to a fixed base (DS register). It is also used for string instructions, as it points to the source character.

DI (Destination Index): Like SI, it points to memory and presents an offset relative to a fixed base (DS or ES). It is also used for string instructions, pointing to the destination.

SP and BP pointers (Stack Pointer and Base Pointer):
These point to the stack area (a memory area managed in Last In, First Out - LIFO order). They have offsets relative to the stack segment (SS register). The BP register plays a role similar to BX but is usually used with the stack segment.

The Stack Pointer **(SP)** allows for pointing to the stack to store data or addresses based on the LIFO principle (Last In, First Out).

4.2.3 The status register (indicator)

The status register is used to store the state of certain operations performed by the processor. For example, the Sign Flag (**SF**) is set to 1 if the last operation resulted in a negative result, and it's set to 0 if the result is positive or zero.

Note: Flags are indicators that signify a specific condition resulting from an arithmetic or logical operation. Each flag is manipulated individually by specific instructions.

The 8086 status register is formed by the following bits:

Figure 2.5. Status Register

×: To denote an unused bit.

CF (Carry Flag): This indicator is set to **1** when there is a carry in the 8 or 16-bit result. It plays a role in addition (carry) and subtraction (borrow) operations on natural numbers.

CF = 1 if there is a carry after adding or subtracting the most significant bit of the operands.

PF (Parity Flag : drapeau de Parité) : si le résultat de l'opération contient un nombre pair de 1, cet indicateur est positionné à **1**, sinon **0**.

PF (Parity Flag): If the result of the operation contains an even number of **1**, this indicator is set to **1**; otherwise, it's set to **0**.

AF (Auxiliary Carry Flag): This bit is set to **1** if there is a carry from the low-order quarter (4 bits) to the high-order quarter.

ZF (Zero Flag): This indicator is set to **1** when the result of an operation is equal to zero. When a subtraction (or comparison) has just been performed, **ZF = 1** indicates that the two operands were equal. Otherwise, **ZF** is set to **0**.

SF (Sign Flag): This indicator is set to **1** if the last operation produced a negative result, and it's set to **0** if the result is positive or zero. **SF** is useful when working with signed integers because the most significant bit indicates the sign of the result.

OF (Overflow Flag): If there is an arithmetic overflow, this bit is set to **1**, meaning the result of an operation exceeds the capacity of the operand (register or memory location); otherwise, it's set to **0**.

DF (Direction Flag): Used by string processing instructions to auto-increment or auto-decrement the **SI** and **DI** registers.

IF (Interrupt Flag): To mask external interrupts, this bit is set to **0**. Otherwise (**IF = 1**), the microprocessor recognizes external interrupts.

TF (Trap Flag): Used to execute the program step by step, causing the microprocessor to execute instructions one at a time.

Notes:

The **IF**, **DF**, and **TF** bits are control indicators that allow the modification of the microprocessor's behavior and are set by the programmer.

5 Definition and Operating Principle of the Stack (Stack Segment):

A stack is a collection of reserved memory locations for data stacking, and its essential characteristic is that the first element introduced into the stack is always at the bottom, and the last element introduced is always at the top (**L**ast **I**n, **F**irst **O**ut - LIFO). The stack is **organized in 16-bit words**.

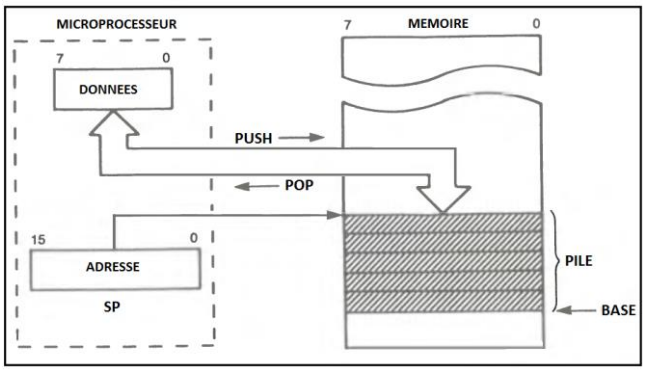


Figure 2.6. Diagram of the two stack manipulation instructions: PUSH and POP.

Operation: In normal use, the stack can be accessed through two instructions, which are **PUSH** and **POP** as depicted in the figure above.

SP (stack pointer) is the register that points to the top of the stack, i.e., the last occupied word of the stack.

The PUSH instruction saves (puts) an element (16 bits) at the top of the stack, and it works as follows:

1) SP ← SP - 2
2) SS:[SP] ← operand (16 bits)

SS:[SP]: Address of the top of the stack

Instruction	Opérande (16 bits)	Description de l'instruction	Indicateurs
PUSH	**Reg** **SReg** **Mem**	**Store 16 bits operand in the stack Segment.**	**Does not affect the flags**

SReg : to designate a segment register, namely CS, DS, ES, and SS.

L'instruction POP fournie (transfère) le mot mémoire du sommet de la pile à l'opérande de destination, elle fonctionne comme suit **:**

1) **Operand** (16 bits) **← SS:[SP]**
2) **SP ← SP + 2**

"SReg: To designate a segment register, namely CS, DS, ES, and SS.

The POP instruction provided (transfers) the memory word from the top of the stack to the destination operand, and it operates as follows:

1) **Operand (16 bits) ← SS:[SP]**
2) **SP ← SP + 2**

Instruction	Operand (16 bits)	Instruction description	Flags
POP	**Reg** **SReg** **Mem**	**Get 16 bits operand from the stack Segment.** This instruction retrieves a 16-bit value from the stack and stores it in the operand.	**Does not affect the flags**

Example :

PUSH AX ; Push the AX register onto the stack segment
PUSH BX ; Push the BX register onto the stack segment
PUSH [1100H] ; Push the value from memory location 1100H-1101H onto the stack
POP [1100H] ; Pop in reverse order of pushing into memory
POP BX
POP AX

Note: The value of SP must be initialized by the main program before the stack can be used.

Table 2.1 Instructions for saving and restoring registers in the stack segment.

Instruction (no operand)	Instruction description	Flags
PUSHF	**Store flags register in the stack Segment.**	**[SP] ← Status register** **SP ← SP - 2**
POPF	**Get flags register from the stack.**	**Status register ← [SP]** **SP ← SP +2**
PUSHA	**Push all general purpose registers AX, CX, DX, BX, SP, BP, SI, DI in the stack.**	**PUSH AX** **PUSH CX** **PUSH DX** **PUSH BX** **PUSH SP** **PUSH BP** **PUSH SI** **PUSH DI**
POPA	**Pop all general purpose registers DI, SI, BP, SP, BX, DX, CX, AX from the stack.**	**POP DI** **POP SI** **POP BP** **POP xx (Ignored SP value)** **POP BX** **POP DX** **POP CX** **POP AX**

6 General Execution Cycle of an Instruction

The processing of an instruction can be divided into four phases:

Phase 1: Instruction Fetch
Phase 2: Instruction Decode and Operand Fetch
Phase 3: Instruction Execution
Phase 4: Result Storage, if applicable

6.1 Instruction Fetch:

The content of the Instruction Pointer (**IP**) register is placed on the address bus and sent to memory. The control bus then generates a memory read signal. When memory receives the read signal, the data it contains at the specified address is placed on the data bus. The microprocessor then reads the information from the data bus and stores it in an internal register called the Instruction Register (**IR**). The information read by the microprocessor is an instruction. It can be said that the instruction has been fetched from memory. Figure 2.7 illustrates this process.

An **incrementer** is connected to the Instruction Pointer (**IP**), and it is used to indicate the address of the next instruction to be executed. Instructions are fetched sequentially during program execution.

6.2 Instruction Decode:

Once the instruction is in the **IR register**, the BIU decodes it and generates the appropriate sequence of internal and external signals for its execution. It takes some time, typically one clock cycle, for the microprocessor to decode an instruction and determine the corresponding action.

6.3 Instruction Execution:

As soon as the CPU decodes the instruction, it performs a series of tasks dictated by the instruction. Some instructions involve fewer tasks compared to others, so their execution time is short. The speed of executing an instruction is generally expressed by the total number of clock cycles required for its execution. The number of cycles depends on the **complexity of the instruction** and the **addressing mode used**.

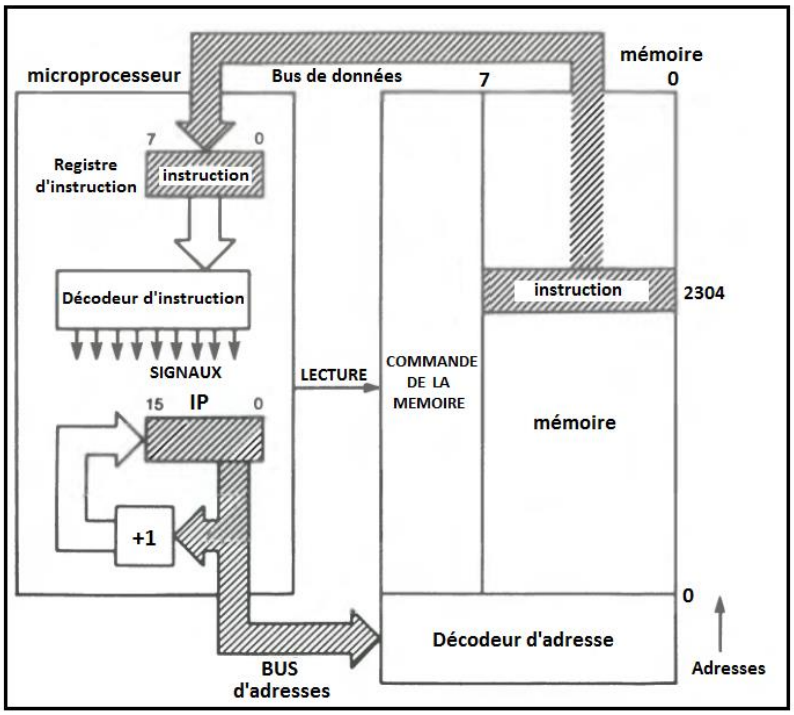


Figure 2.7. Diagram of Instruction Fetch from Memory.

Example :

Tableau 2.2. Description of the ADD instruction

ADD	**ADD destination, source** **Addition**		**Drapeaux** O D I T S Z A P C X X X X X X
Opérandes	**Cycles horloge**	**Octets**	**Exemple de programme**
register, register	3	2	ADD CX, DX
register, memory	9 + EA	2-4	ADD DI, [BX].ALPHA
memory, register	16 + EA	2-4	ADD TEMP, CL
register, immediate	4	3-4	ADD CL, 2
memory, immediate	17 + EA	3-6	ADD ALPHA, 2
accumulator, immediate	4	2-3	ADD AX, 200

Tableau 2.3. Description of the MUL instruction

MUL	**MUL source** **Multiplication non signée**		**Drapeaux** O D I T S Z A P C X U U U U X
Opérandes	**Cycles horloge**	**Octets**	**Exemple de programme**
reg8	70-77	2	MUL BL
reg16	118-133	2	MUL CX
mem8	(76-83) + EA	2-4	MUL MONTH [SI]
mem16	(124-139) + EA	2-4	MUL BAUD_RATE

EA: to refer to the **E**ffective **A**ddress of the data in memory.

As you can see, the execution time taken by the microprocessor to execute the **MUL** instruction (**70-77 clock cycles**) is much longer than that of the **ADD** instruction (**3 clock cycles**). (See the two tables above).

7 Instruction Encoding

Instructions and their operands (parameters) are stored in main memory. The total size of an instruction depends on the type of instruction and the type of operand. An instruction consists of two fields:

- **Operation code** (Opcode) tells the processor which instruction to execute.
- **Operand field** contains the data or a reference to data in memory (its address).

Instruction	Operation code	Operand field
MOV AX, BX	**8B H (1 octet)**	**C3 H (1 octet)**

Each instruction begins with the **opcode** that determines the nature of the instruction, followed by the operand field indicating the type of operands, their location, or an immediate value (literal).

Exemple :

Instructions	**Opcode**	**Operand field**	**Size (bytes)**	**Code machine**
MOV AX, BX	8B H	C3 H	2	1000 1011 1100 0011
MOV DS, AX	8E H	D8 H	2	1000 1110 1101 1000
MOV AX, 5H	8B H	0500 H	3	1000 1011 0000 0101 0000 0000
MOV AL, temper	A0 H	0E01 H	3	1010 0000 0000 1110 0000 0001

Machine language, or **machine code**, is the sequence of bits interpreted by the computer's microprocessor when executing a computer program. **It is the only language that the processor can process**.

8 Enhancements to the basic architecture

All the improvements in microprocessors are aimed at **reducing program execution time**. Another way to increase the processing power of a microprocessor is to reduce the average number of clock cycles required

for executing an instruction. Another solution is to employ a microprocessor architecture that reduces the number of cycles per instruction.

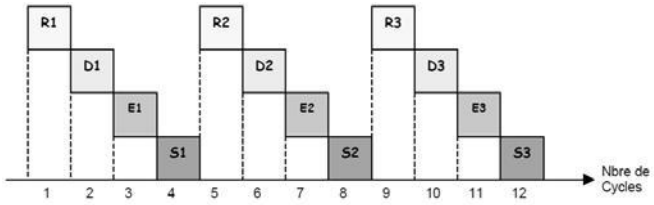
8.1 Pipeline Architecture

The execution of an instruction is broken down into a series of stages, with each stage corresponding to the use of one of the microprocessor's functions (fetching, decoding, execution). When an instruction is in one of the stages, the components associated with the other stages are not in use. Therefore, the operation of a simple microprocessor is not efficient.

Pipeline architecture improves the efficiency of the microprocessor. When the first stage of executing an instruction is completed, the instruction moves to the second stage of execution, and the first phase of the next instruction's execution begins. A pipeline machine is characterized by the number of stages used for the execution of an instruction, or the number of pipeline stages.

Example of the 4-phase execution of an instruction:

Classic Model:



Pipeline Model :

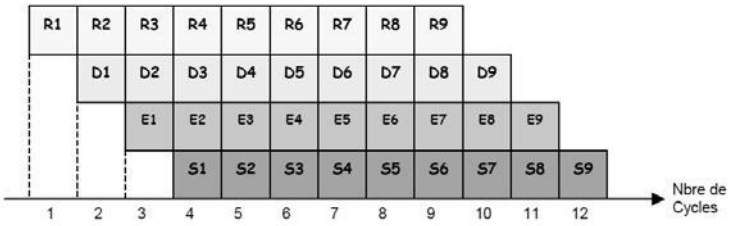


Figure 2.8. Comparison between the classic execution model and the pipeline execution model

8.2 Memory Cache Concept

The performance gap between the microprocessor and memory continues to widen. Memory components benefit from the same technological advancements as microprocessors, but decoding addresses and reading/writing data are challenging steps to accelerate. Memory is no longer able to deliver information as quickly as the processor. There is, therefore, an access latency between these two components.

One of the solutions used to mask this latency is to place a very fast memory between the microprocessor and the main memory. It is called "**memory cache**." This compensates for the relative slowness of memory by allowing the microprocessor to acquire data at its own speed. Initially, this memory was integrated outside the microprocessor, but it is now an integral part of the microprocessor and is even available in multiple levels.

The principle of cache is very simple: the microprocessor is unaware of its presence and sends all its requests to it as if it were the main memory:

Either the required data or instruction is present in the cache, and it is then sent directly to the microprocessor. This is referred to as a cache hit.

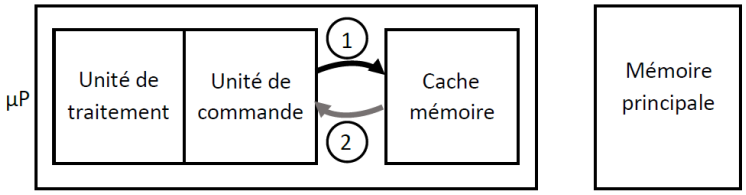


Figure 2.9. Cache Hit

Either the data or instruction is not in the cache, and the cache controller then sends a request to the main memory. Once the information is retrieved, it is sent back to the microprocessor while also being stored in the cache. This is referred to as a cache miss."

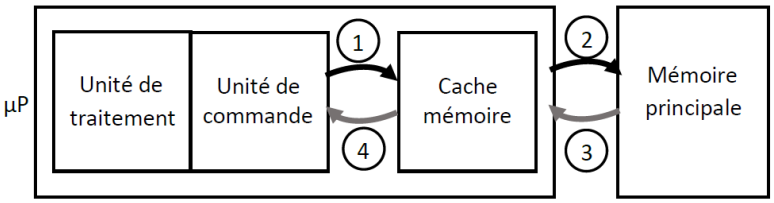


Figure 2.10. Figure 2.10. Cache Miss.