University M'hamed BOUGARA-Boumerdes

Faculty of Technology

Department of Electrical Systems Engineering

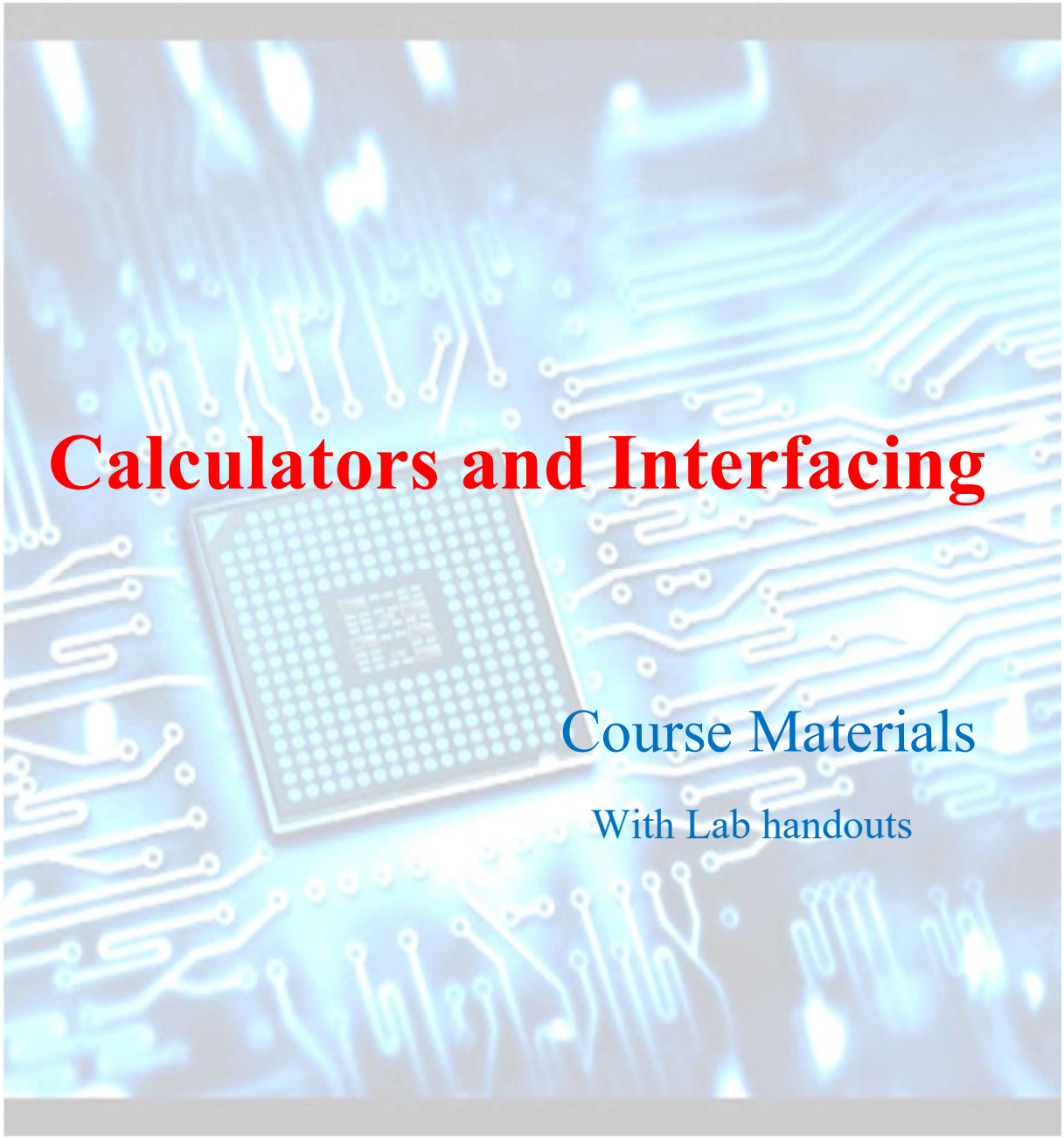
Calculators and Interfacing

Course Materials

With Lab handouts

Karim BAICHE

2024



Calculators and Interfacing

Course Materials

With Lab handouts

Karim BAICHE

MCA, Department of Electrical Systems Engineering

Faculty of Technology

University M'hamed BOUGARA-Boumerdes

Year - 2024

Summary

Introduction 3
Chapter 1: General and historical information on Computer Science 2
1.1-Introduction 2
1.2- A bit of history 4
1.2.1- Introduction 4
1.2.2- Mechanical calculation 5
1.2.3- Electromechanical calculation 6
Chapter 2: Basic Architecture 11
2.1- Von Neumann model 11
2.2- The Central Processing Unit 12
2.3- The Central Memory (Main Memory) 12
2.4- I/O interfaces 12
2.5- Buses 13
Chapter 3: The Processor (Central Processing Unit, CPU) 14
3.1- Introduction 14
3.2- Basic architecture of a microprocessor 14
3.2.1- The Control Unit 15
3.2.2- The Processing Unit 15
3.2.3-Execution Cycle of a Statement/Instruction 16
3.2.4- Instruction Set 18
3.3- Special Processors 27
Chapter 4: The Intel 8086 Microprocessor 29
4.1- Introduction 29
4.2- Internal Architecture of the intel 8086 Microprocessor 29
4.2.1- Description of the intel 8086 29
4.2.2- The 8086 Microprocessor Registers 29
4.3- Representation and coding of instructions 31
4.4 8086 Instructions 32
4.4.1 Definition 32
4.4.2- Instruction Description Template 32
4.4.3- Transfer instructions 34
4.4.4- Increment, decrement 35
4.4.5 Opposite of a number 36
4.4.6 Arithmetic Instructions 36
4.4.7 Boolean and logical instructions 40

4.4.8 Assembler tests 43
4.4.9 The Stack 47
4.4.10 Input-Output Instructions: 48
4.4.11 Shift and Rotate Instructions 48
Chapter 5: Memories 52
5.1- Introduction 52
5.2- Organizing a memory 52
5.3- Characteristics of a memory 53
5.3.1- Capacity 53
5.3.2- The format of the data 54
5.3.3- Access time 54
5.3.4- Le temps de cycle 54
5.3.5- Throughput 54
5.3.6- Volatility 54
5.3.7- Modes of access 54
5.4- Different types of memory 55
5.4.1- Random Access Memory (RAM) 55
5.4.2- Criteria for choosing between SRAM and DRAM 57
5.4.3- Read Only Memories 57
Chapter 6: Input/Output Interfaces 64
6.1- Introduction 64
6.2- The I/O Interface 64
6.3- Data Exchange Techniques 65
6.3.1- Polling 65
6.3.2- Interruption 65
6.3.4- Direct Exchange to Memory (DMA) 67
6.4- Types of Links 68
6.4.1- Parallel Link 68
6.4.2- Serial Link 69
6.5- Architecture of a PC 71
6.5.1- The chipset 71
6.5.2- BIOS (Basic Input Ouput Service) 72
6.5.3- The Clock 72
6.5.4- Connection ports 72
6.5.5- The socket 72
Lab Handouts 73
Biblipgraphy .....

Introduction

This course of "Calculators/Computers and Interfacing" or "Computer Architecture" is aimed at students in the field ST (Science and Technology), electrical engineering (Telecommunication, Electronics, ...) as well as students of MI (Mathematics and Computer Science).

It aims to give a global idea, for these students, about computer machines, from the architectural point of view as well as their operating principle.

It is very clear that nowadays, computer systems or intelligent systems are invading us. They are everywhere in our daily lives, telephone, TV, washing machine and even the house itself (smart home). For this, it is necessary for anyone working in this field or using microprocessor systems to have a clear idea about the architecture and operation of their systems.

The document is divided into 6 chapters. Chapter 1 presents some notions of computer science as well as the historical evolution of these systems. Chapter 2, gives an idea of the basic architecture of computer machines. Chapter 3 and 4, are complementary, the first presents the architecture and operation of the processor in a general way as for the 4^{e} it presents in detail the Intel 8086 processor. Chapter 5 presents the different types of memories and their operating principles. And to close, the 6th chapter details the different modes of communication with external devices.

Chapter 1: General and historical information on Computer Science

1.1-Introduction

Initially a computer was a digital calculator. It is now an information processing machine. It is able to acquire and store information, perform processing and retrieve information.

Computer science has become:

« THE SCIENCE OF INFORMATION PROCESSING»

A computer can be cut into functional blocks. Information processing is done at the level of a processor. The actions that it must perform are defined by instructions. To be accessible to the processor the data to be processed and the instructions must be stored in a memory. The processor and memory are connected by a bus. In addition, the user must be able to provide the computer with data and instructions to follow, as well as the results. Inputs and Ouputs devices are therefore required.

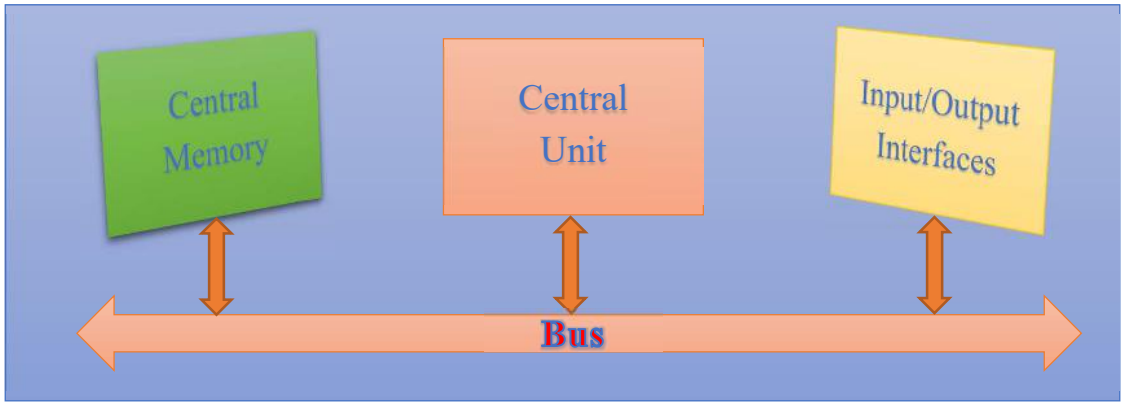


Figure 1:Components of a computer

Each functional block can itself be described by a set of functional units. This is referred to:

- Processor architecture,
- Memory architecture, etc.
- Architecture of a network with multiple computers.

This shows the emergence of the notion of levels of representation. At each level an object considered is described as a set of interconnected blocks. At the next level each of these blocks is in turn described as a set of interconnected blocks and so on.

This hierarchical breakdown into levels depends on the needs of the design or the tools used: it is neither fixed nor unique.

We can, for example, go down to the level of logic gates and even to the level of the transistors that make up these gates.

The architecture of a computer is the representation of its functional units and their interconnections.

The choice of an architecture is always the result of a compromise:

- Between performance and cost;
- Between efficiency and ease of construction;
- Between overall performance and ease of programming;
- etc...

Each solution adopted for a given machine, at a given moment and for a given market, can very quickly be called into question by a new technological progress. Similarly, architectures proposed at one time, then abandoned for technical or economic reasons, may one day be used again.

Whatever its size (mini, micro, super, hyper, etc.) we always find in a computer the same functional blocks:

- One or more processing units;
- Memory;
- One or more hard drives, constituting permanent memory;
- Communication devices with the operator: a screen, a keyboard, very often a mouse;
- External communication devices such as a printer;
- Devices for archiving: magnetic tape, optical disk;
- etc.

Connected by buses, links or networks.

Computing is also embedded in a large number of everyday devices, as varied as

- Washing machines,
- Beverage dispensers,
- Cars
- Or bank cards.
- Etc...

If, compared to a conventional computer, the tasks to be carried out are not as versatile and if the constraints are different, there are however the same bricks.

It is important to realize that:

- Hardware and software are inseparable, especially at the design level of any computer system.
- As far as the hardware is concerned, we detail some functional blocks.
- As far as software is concerned, we will limit ourselves to a few simple considerations about machine language.

1.2- A bit of history

1.2.1- Introduction

To summarize the history of computer science or computers, it is often necessary to schematize. For each invention, we usually retain only one name. But often this invention concretizes a collective approach that has been spread over more or less time. Very often also predecessors have been forgotten. Man has always needed means of calculation. Let us cite as an example the word itself, whose etymology *calculi* means pebbles in Latin (used on the abacuses of the Romans).

Figure 2: Abacuses of the Romans

To be able to calculate we needed numbering. As an example, we cite the Roman numeration: MDCCCLXXIIIRomain=187310. Or the decimal numbering linked to the technology of the first pocket calculator: the hand (digiti). So, Digital Computing then meant counting on one's fingers.

A very important innovation was the use of positional notation, which gives different values to numerical symbols according to their position in the written number. This positional notation is only possible with a symbol for zero. Thanks to the symbol 0, it became possible to differentiate 11, 101 and 1001 without resorting to additional symbols (MDCCCLXXIIIRomain).

This notation was introduced to Europe through the Arabs, with Islam extending from the borders of China to Spain.

The so-called Arabic system had been developed in India about 300 BC. This introduction was made thanks in particular to the translation, around 820, of the works of the Baghdad mathematician Al-Khuwarizmi, whose title of one of the books (al-jabr) is at the origin of the word algebra. The first documents attesting to the use of the Arabic system in Europe date from 976, but it was not until the fourteenth century that it completely replaced the Roman numbering. Not only did the writing of numbers become more compact, but written calculations were greatly simplified.

1.2.2- Mechanical calculation

- XVIIth century: appearance of the first mechanical calculation systems, based on cogwheels (gearweels).
- 1614, when John Napier (or Neper), Scottish mathematician, invented the first logarithms. He thus reduced the operations of multiplication and division to simple additions or subtractions.

Figure 3: The Neperian

- 1622: William Oughtred improved Neper's work by giving more precision.
- In 1623, Wilhelm Schickard invented for Kepler a "calculating clock" for calculating ephemeris.
- In 1642, at the age of 19, Blaise Pascal built, to help his father commissioner for taxation, an "arithmetic machine" capable of processing six-digit additions and subtractions.
- The Pascaline is considered the first self-retaining adder.

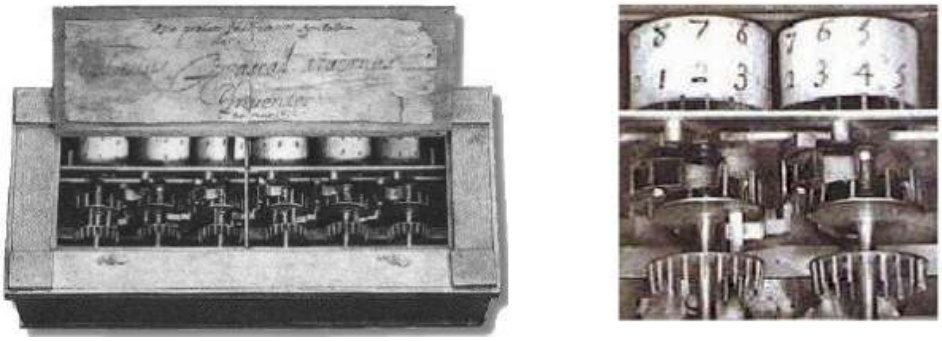


Figure 4: Pascaline

- In 1673, Gottfried Leibniz improved Pascaline by automating the repetitive additions and subtractions necessary for multiplication and division.
- Leibniz also invented the binary system and showed the simplicity of binary arithmetic.
- In 1728, the French mechanic Falcon built the first loom and ordered its operation with a wooden board pierced with holes.
- It is the first machine controlled by a program.
- 1820-1830, an English mathematician, Charles Babbage, brought together calculating machines and control systems in order to perform complex calculations requiring the execution in sequence of several arithmetic operations.
- In particular, this machine defined the principle of the sequence of successive iterations for the realization of an operation, named algorithm in honor of the Arab mathematician Al-Khuwarizmi.
- In 1854 George Boole proposed his mathematical formulation of logical propositions which applied to the binary system is the basis of the functioning of computers.

1.2.3- Electromechanical calculation

- In 1890, Hermann Hollerith built a statistical calculator that was used for the U.S. Census. It was an electromechanical machine that performed better than mechanical machines. On this occasion he developed the punch card and invented the information coding system that bears his name. The presence or absence of a hole was detected by means of needles that passed through the holes and each closed an electrical circuit by dipping in a bucket of mercury.
- Hollerith founded the Tabulating Machine Company in 1896 to produce his maps and mechanographic machines.

- In 1924, it became the International Business Machines Corporation: IBM.
- In 1904: John Fleming invents the diode (the first vacuum tube)

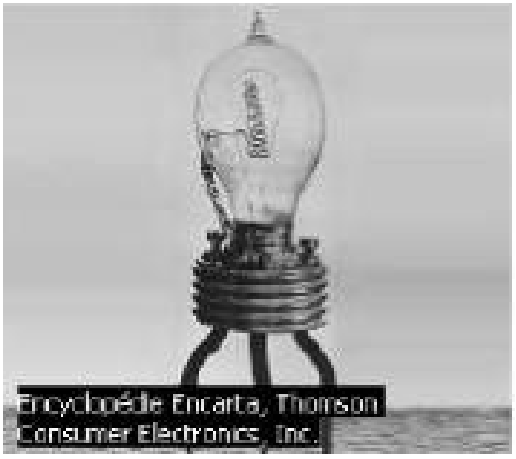


Figure 5: Vacuum tube

- In 1938, Konrad Zuse created with modest means a mechanical programmable binary computer, the Z1.
- In 1939, he perfected his machine by replacing some of the mechanical parts with electromechanical relays (Z2).
- In 1941 the Z3 and Z4 were born and were used for aeronautical calculations.
- In 1939, John Atanasoff and Clifford Berry made a 16-bit binary adder. They were the first to use vacuum tubes.
- In 1941 IBM and Harward, jointly developed one of the last electromechanical computers the Mark 1 or ASCC (Automatic Sequence Controlled Calculator). It was a huge 5-ton machine, covering 25 m2 and consuming 25 kW. It had 3000 relays and 760000 mechanical parts. The program was read on a strip of paper, the data from a second strip of paper or a card reader.

- In 1945, an insect (bug) jammed a relay, causing a malfunction of this analytical machine, hence the computer term "Bug" which means a malfunction in a program.

2. The Age of Electronics

A. 1st generation (1940-1960), use of vacuum tubes

- 1940: invention of the printed circuit board (wafer with tracks to connect components)

- Also in 1941, John Atanasoff and Clifford Berry built the first binary tube computer: the ABC (Atanasoff-Berry Computer). It had a memory of 60 words of 50 bits and an arithmetic and logical unit. Although the program was not stored in memory, the ABC is often considered the first real computer.

- Early 1945: ENIAC (Electronic Numerical Integrator And Calculator), 1st programmable electronic computer but requiring to reconnect hundreds of cables for each calculation because its internal memory was too small;

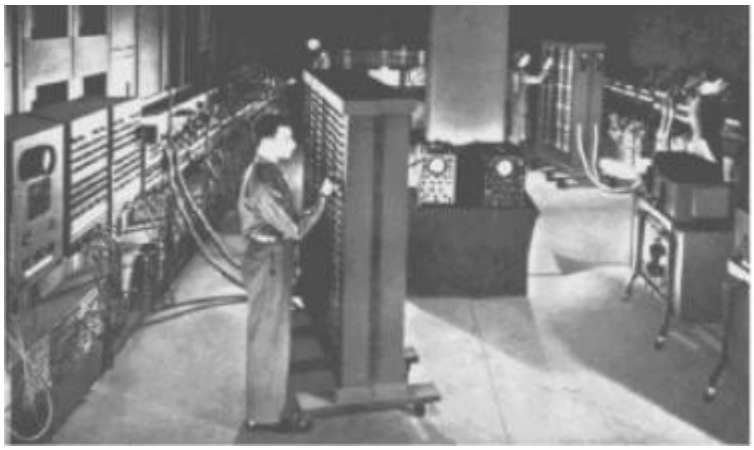


Figure 6: ENIAC

- It consisted of 19000 tubes, 1500 relays, consumed 170 kW, weighed 30 tons and covered an area of 72 m2. It was about 500 times faster than the Mark 1 (about 330 multiplications per second)
- However, its programming was done using plugs to be plugged into a table of connections.
- The programming work could take several days.
- At the end of the same year, John von Neumann, a consultant on ENIAC, proposed to encode the program in digital form and save it in memory, with greater flexibility and speed. It thus laid the foundations for the architecture of the modern computer.
- In 1948, William Shockley, John Bardeen and Walter Brattain invented the bipolar transistor.
- It quickly replaced the lamps that brought reliability and speed to second-generation computers.
- Size and consumption decreased significantly.

Figure 7: Semiconductor transistor

B. 2nd generation (1960-1970), use of transistors

- The first computer to use transistors was the TRADIC in 1955.
- At the same time, IBM marketed the first hard disk (5 disks of 61 cm in diameter for 5 MB).
- Ferrite torus memories were also available.
- DEC's PDP-8 was the first minicomputer to be distributed in large series (50,000 copies).
- 1st DBMS (database management system);
- 1st integrated circuits;
- 1st programming languages (1960: Lisp, Cobol, Fortran, 1964: Basic)

C. 3rd generation (1970-1980), use of integrated circuits

- Corresponds to the use of integrated circuits.
- 1971, the Intel 4004, the first 4-bit microprocessor, the first integrated circuit incorporating compute unit, memory and input-output management. It had 2300 transistors.

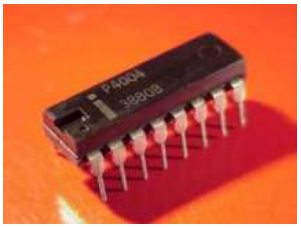


Figure 8: Intel 4004 Microprocessor

- 1972, the 8008 (8 bits, 200 KHz, 3500 transistors). The first microcomputer, the Micral N, was built in 1973 by a French company R2E.
- 1st multi-user operating systems:
    1. Multics (1969)
    2. Unix (1972)
    3. 1971: Arpanet (ancestor of the internet)

- 1974: François Moreno invents the smart card
- 1965: Gordon Moore notes that "the number of transistors that can be integrated on an integrated circuit chip doubles every 18 (to 24) months". **Moore**'s law ("experimental law/observation", it has always been true): the power of new microprocessors and the capacity of new memories double every 18 months at most (between 12 and 18 months)

D. 4th generation (1980-1990), use of large-scale integration

- Chip incorporating hundreds of thousands of transistors

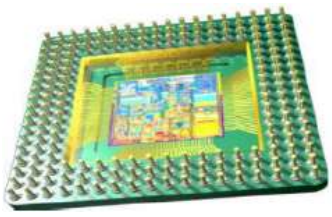


Figure 9: Large scale integrated circuit

- Personal computers;
- Peripherals (mouse, CD-ROM, ...);
- Internet
- 1980: A branch of IBM adopts MS-DOS (developed, discontinued, and sold to MicroSoft by another branch of IBM). Microsoft monopolizes the software market on the best-selling machine (PC).
- 1991: Linus Torvalds creates Linux by rewriting/lightening the Unix kernel.

E. 5th generation (1990-...), Parallelism & Internet & Increasingly large integration

- Parallelism (in the microprocessor, several microprocessors, ...)
- Semiconductor memories
- Beginning of the merger of computing, telecommunications and multimedia, WWW: World Wide Web
- Increasing integration
- Very large-scale integration: (more than 1 billion transistors with 3 nm lithography).
- Wireless network (Wifi, 5G, etc...)

Chapter 2: Basic Architecture

2.1- Von Neumann model

To process information, a microprocessor alone is not enough, it must be inserted into a system with a minimum of programmed information processing.

John Von Neumann is at the origin of a model of a universal machine of programmed information processing **(1946).**

This architecture serves as the basis for most microprocessor systems today.

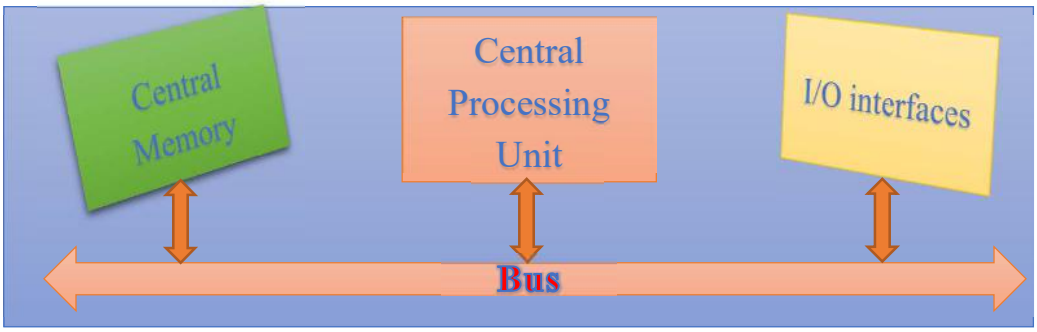


Figure 10 : Architecture of Von Neumann

According to this way of seeing, two architectures have been developed. That of Von Neumann which consists in putting the programs (codes) as well as the data on the same memory (a single address bus) but separate into segments (CS: Code segment, DS: Data segment) and the other of Hardvard which separates between the program memory and the data memory (two address buses).

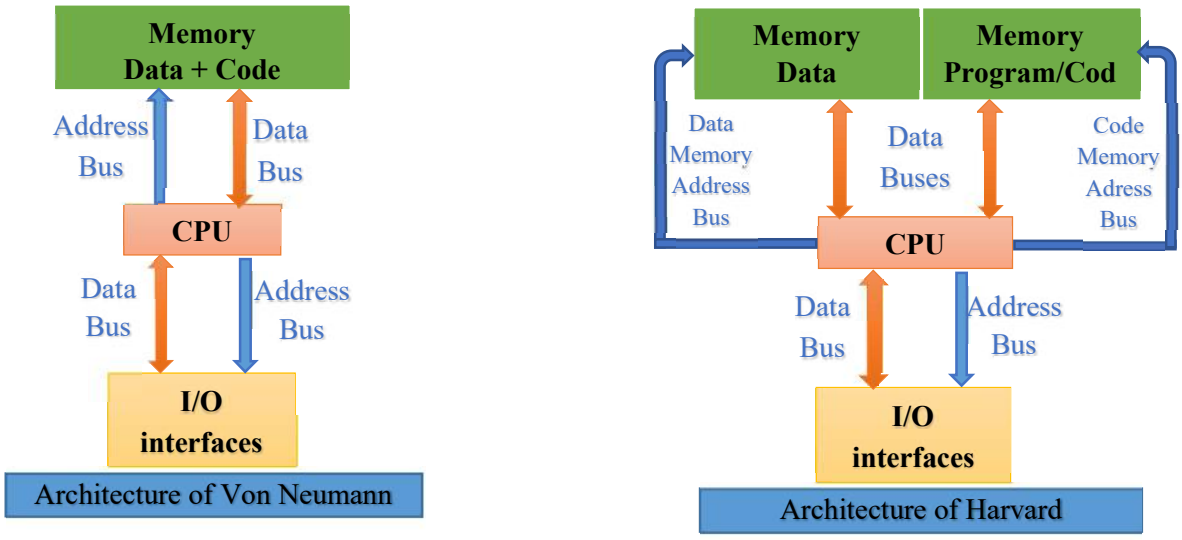


Figure 11: Architecture of Von Neumann VS Architecture of Harvard

2.2- The Central Processing Unit

It is composed of the microprocessor which is responsible for interpreting and executing the instructions of a program, reading or saving the results in memory and communicating with the exchange units.

All activities of the microprocessor are clocked by a clock. The microprocessor is characterized by:

1. Its clock frequency: in MHz or GHz
2. The number of instructions per second it is able to execute in MIPS (million instructions per second)
3. The size of the data it is able to process in bits

2.3- The Central Memory (Main Memory)

It contains the instructions for the running program or programs and the data associated with that program.

Physically, it often breaks down into:

1. A read-only memory (**ROM** = Read Only Memory) responsible for storing a program. It is a read-only memory.

2. A random-access memory (RAM = Random Access Memory) responsible for storing intermediate data or calculation results. You can read or write data in it, this data is lost when power is turned off.

Note: Hard drives, flash drives, CDROMs, etc. are storage devices and are considered secondary memories.

2.4- I/O interfaces

They ensure communication between the microprocessor and peripherals:

- Sensor
- Keyboard
- Monitor or display,
- Printer
- Modem
- …

2.5- Buses

A bus is a set of wires that ensures the transmission of the same type of information. There are three types of buses carrying information in parallel in a programmed information processing system:

Data Bus: bidirectional that ensures the transfer of information between the microprocessor and its environment, and vice versa. Its number of lines is equal to the processing capacity of the microprocessor (8, 16, 32 or 64 bits).

Address Bus: unidirectional that allows the selection of information to be processed in a *memory space* (or *addressable space*) that can have 2^n locations, with n = number of wires of the address bus.

Control Bus: consisting of a few wires that ensure the synchronization of information flows on the data and address buses.

Note: When a component is not selected, its outputs are set to the "high impedance" state so as not to disturb the data circulating on the bus (it has a very high output impedance = open circuit).

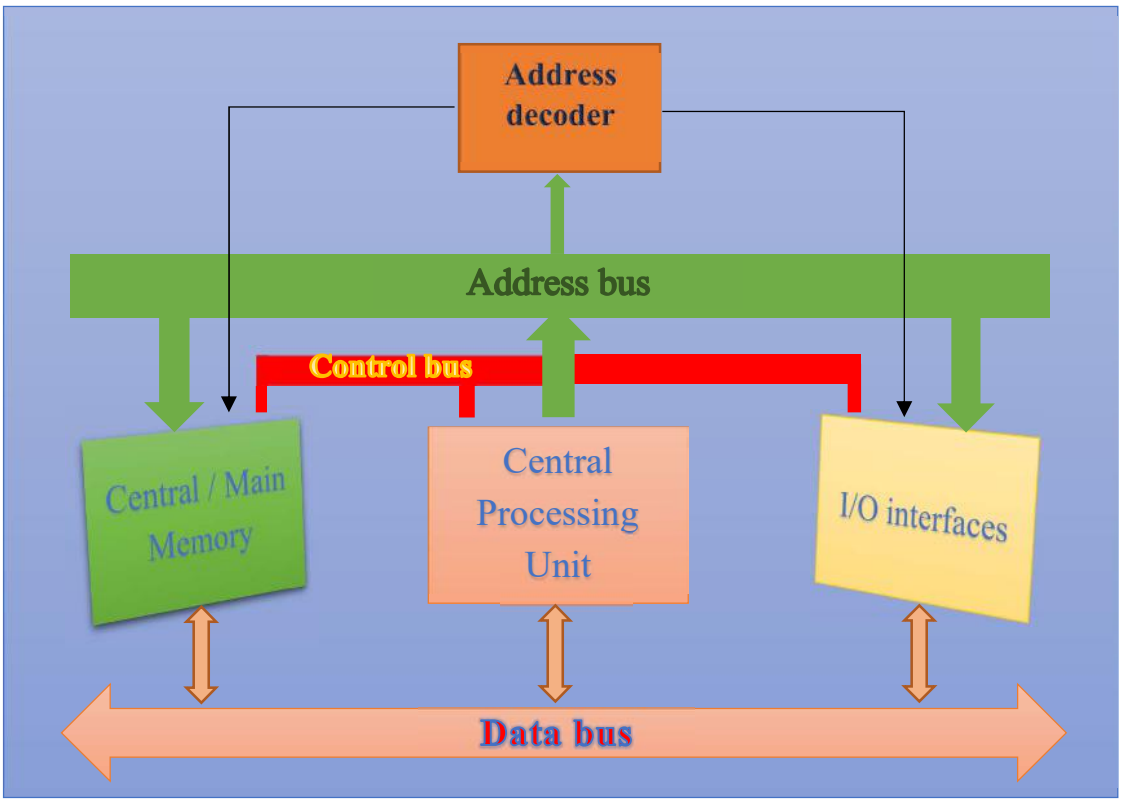


Figure 12: The different system buses

Chapter 3: The Processor (Central Processing Unit, CPU)

3.1- Introduction

A microprocessor is a complex integrated circuit (a few square millimeters) characterized by:

- A very large integration (a few billion)
- Endowed with the ability to interpret and execute the instructions of a program.

It is responsible for:

- Organize the tasks specified by the program
- Ensure their execution.

It is the brain of the system!

3.2- Basic architecture of a microprocessor

A microprocessor is built around two main elements:

- One control unit
- A processing unit

associated with registers responsible for storing the various information to be processed.

These three elements are linked together by internal buses allowing the exchange of information.

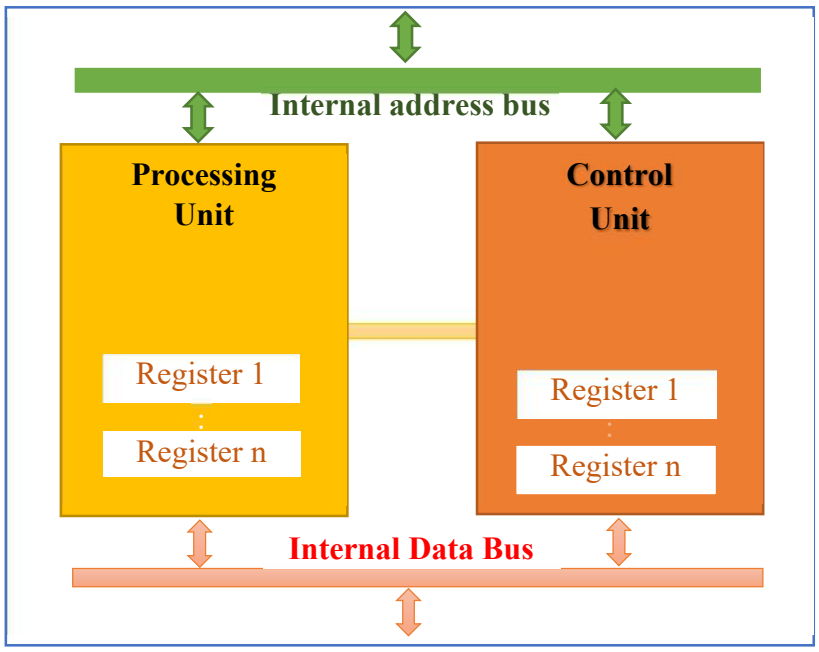


Figure 13: CPU Architecture

3.2.1- The Control Unit

It makes it possible to sequence the sequence of instructions. It performs:

- The in-memory search of the instruction.
- The decoding of it
- Ensure its execution
- Preparation of the following instruction.

In order to carry out its task, it is composed of:

Program counter (PC): a register whose contents are initialized with the address of the first instruction of the program. It always contains the address of the statement to be executed.

Instruction Register (IR): arranges the statement to be executed

Instruction decoder: Decodes the instruction to be executed and generates the appropriate codes.

Command/Control Logic Block (or Sequencer):

- Organizes the execution of instructions at the beat of a clock.
- Elaborates all internal or external synchronization signals (control bus) of the microprocessor according to the various control signals from the instruction decoder or the status register for example.
- It is an automaton made either in such a way:
  - Cable
  - Micro-programmed, we then speak of micro-microprocessor.

3.2.2- The Processing Unit

It includes the circuits that provide the processing necessary for the execution of instructions.

The Arithmetic and Logical Unit (ALU): which is a complex circuit that ensures logical functions (AND, OR, Comparison, Shift, etc.) or arithmetic (Addition, subtraction).

Accumulators: which are working registers that are used to store an operand at the beginning of an arithmetic (logical) operation and the result at the end of the operation.

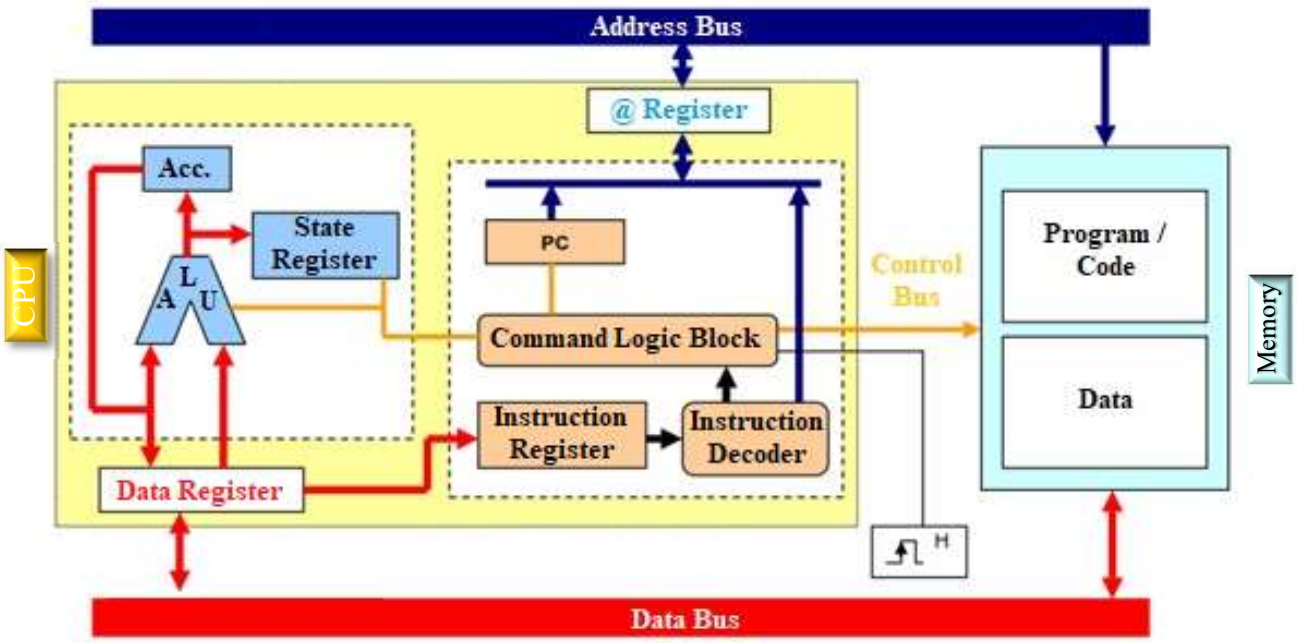


Figure 14: Block diagram of the Microprocessor

3.2.3-Execution Cycle of a Statement/Instruction

The microprocessor only includes a certain number of instructions which are coded in binary. The processing of an instruction can be broken down into three phases.

Phase 1: Search for the instruction to be processed

1. The PC (Program counter): contains the address of the following program instruction. This value is placed on the address bus by the control unit that emits a reading order.
2. After a certain time (memory access time), the contents of the selected memory box are available on the data bus.
3. The instruction is stored in the processor's IR (Instruction Register).

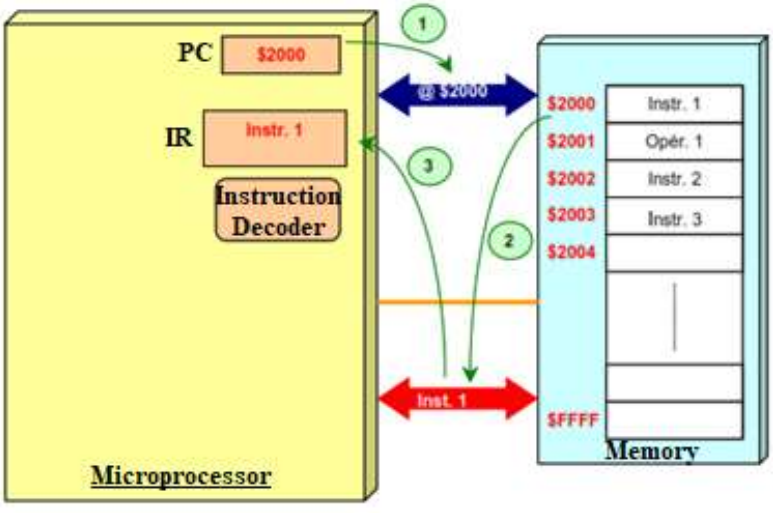


Figure 15: Phase 1, Search for the instruction to process

Phase 2: Decoding the instruction and finding the operand

1. The IR register now contains the first word of the instruction that can be encoded over multiple words.
2. This first word contains:
   - The operating code that defines the nature of the operation to be performed (addition, rotation, ...)
   - And the number of words of the instruction.
3. The control unit transforms the instruction into a sequence of elementary commands needed to process the instruction.
4. If the instruction requires data from memory, the control unit retrieves its value from the data bus.
5. The operand is stored in a register.

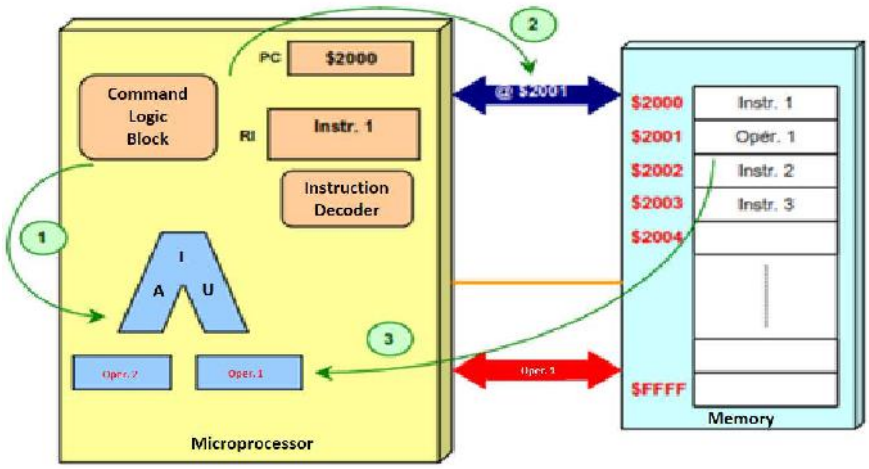


Figure 16: Phase 2, Decoding the instruction and finding the operand

Phase 3: Executing the Instruction

1. The firmware performing the instruction is executed.
2. The flags are positioned (*state register*).
3. The control unit positions the PC for the next instruction.

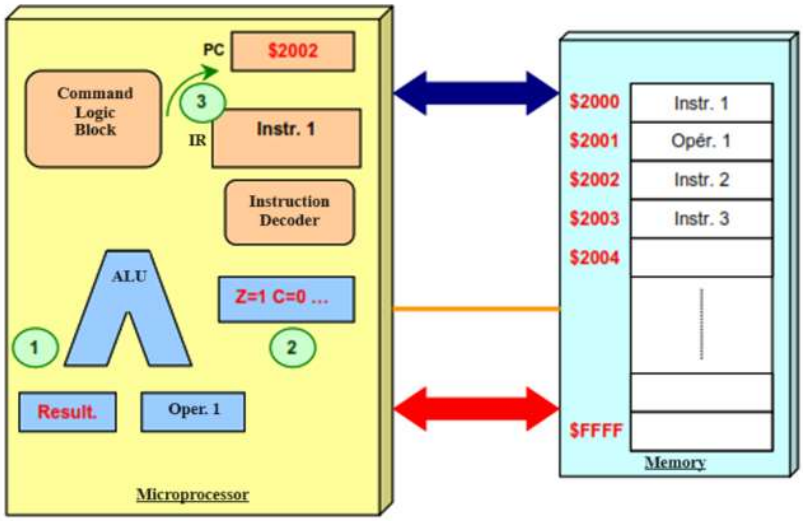


Figure 17: Phase 3, Execution of the statement

3.2.4- Instruction Set

The first step in designing a microprocessor is defining its instruction set. The instruction set describes the set of elementary operations that the microprocessor will be able to perform. It will therefore partly determine the architecture of the microprocessor to be realized and in particular that of the sequencer.

3.2.4.1- Instruction Type

The instructions found in each microprocessor can be classified into 4 groups:

- **Data transfer** to load or save in memory, perform transfers from register to register, etc.
- **Arithmetic operations:** addition, subtraction, division, multiplication, etc.
- **Logical operations:** AND, OR, NO, NAND, comparison, test, etc...
- **Sequence control:** connection, test, etc...

3.2.4.2- Coding

Instructions and their operands (parameters) are stored in main memory. The total size of an instruction (the number of bits needed to represent it) depends on the type of instruction and also on the type of operand.

Each instruction is always encoded on a whole number of bytes to facilitate its decoding. A statement/instruction consists of two fields:

- The instruction code, which tells the processor which instruction to perform
- The operand field that contains the data, or the reference to a data in memory (its address).

Operation Code	Operand Code
1110 0101	1010 0010
E 5	A 2

The number of instructions in the instruction set is directly related to the format of the instruction code. Thus, one byte makes it possible to distinguish a maximum of 256 different instructions.

3.2.4.3- Addressing mode

An addressing mode defines how the microprocessor will access the operand. The different addressing modes depend on the microprocessors but we generally find:

- **Register addressing** where the data contained in a register is processed (Addition of Reg1+REG2)
- **Immediate addressing** where we immediately define the value of the data (Reg Incrementation)
- **Direct addressing (indirect, indexed, ...)** where we process a data in memory:
  - ✓ Direct: Operand code contains the address of the operand
  - ✓ Indirect: Operand code contains the address of the operand address
  - ✓ Indexed: Operand code contains an index that must be added to the index register to have the operand address
  - ✓ …

Note: *Depending on how the data is addressed, a statement will be encoded by one or more bytes.*

3.2.4.4- Execution time

Each instruction requires a certain number of clock cycles to complete. The number of cycles depends on the complexity of the instruction and also on the addressing mode. It takes longer to access the main memory than a processor register.

The duration of a cycle depends on the clock frequency of the sequencer.

3.2.4.5- Programming languages

1- Machine language:

It is a language understood by the microprocessor. It is difficult to master since each instruction is encoded by its own sequence of bits.

In order to facilitate the task of the programmer, different languages have been created more or less evolved.

2- Assembly language:

It is a language "close" to machine language. It is composed by instructions generally quite rudimentary called **mnemonics**. These are essentially operations of :

- Data transfer between registers and outside the microprocessor (memory or peripheral),
- Or arithmetic or logical operations

Each statement represents a different machine code. Each microprocessor may have a different assembler.

3- High-level language:

It is more adapted to humans, and the applications they sought to develop:

- Abstracting from any machine architecture
- These languages allow the expression of algorithms in a form easier to learn, and to dominate (C, Pascal, Java, etc...).
- Each instruction in high-level language will correspond to a succession of instructions in assembly language (machine language).

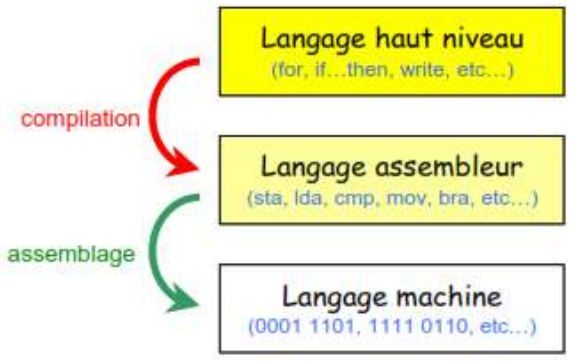


Figure 18: Different levels of programming languages

3.2.4.6-Microprocessor Performance

We can characterize the power of a microprocessor by the number of instructions it is able to process per second. To do this, we define:

1. **CPI** (Cycle Per Instruction) which represents the average number of clock cycles required to execute an instruction for a given microprocessor. It is calculated by applying the arithmetic mean of the microprocessor instruction set, as follow:

$$CPI = \frac{\sum_i \boldsymbol{CI_i} * \boldsymbol{N_i}}{\sum_i \boldsymbol{N_i}}$$

Where, $\boldsymbol{CI_i}$ is the cycle (the number of clock cycles) of some instructions, and $\boldsymbol{N_i}$ is the number of such instructions in the entire Microprocessor instruction set.

2. **MIPS** (Millions of Instructions Per Second) which represents the processing power of the microprocessor.

$$MIPS = \frac{\boldsymbol{F_H}}{\boldsymbol{CPI}}$$

where F_H is the frequency of the Microprocessor in Mhz.

Example: If your microprocessor is clocked at 2.8 GHz ($\boldsymbol{F_H}$ =2.8x10^3 Mhz), and it has the following instruction set detail matrix:

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 10 & 24 & 15 & 8 & 5 \end{bmatrix}$$

where the 1st row is $\boldsymbol{CI_i}$, the cycle (the number of clock cycles of some instructions) and the 2nd row is $\boldsymbol{N_i}$, the number of them in the set. Applying this, we get:

$$CPI = \frac{\sum_i \boldsymbol{CI_i} * \boldsymbol{N_i}}{\sum_i \boldsymbol{N_i}} = \frac{\boldsymbol{1*10+2*24+3*15+4*8+5*5}}{\boldsymbol{10+24+15+8+5}} = \frac{\boldsymbol{160}}{\boldsymbol{62}}$$

$$CPI = \boldsymbol{2.58}$$

having $\boldsymbol{F_H}$ and ***CPI***, we get:

$$MIPS = \frac{\boldsymbol{F_H}}{\boldsymbol{CPI}} = \frac{\boldsymbol{2,8*10^3}}{\boldsymbol{2.58}} = \boldsymbol{1085}$$

So, the power/performance of this microprocessor is 1085 million instructions per second (MIPS). To increase the performance of a microprocessor, we can:

1. Either increase the clock frequency (hardware limitation),
2. Either reduce the CPI (choice of a suitable instruction set).

3.2.4.7-Concept of RISC and CISC architecture

Currently the architecture of microprocessors consists of two main families:

- CISC (***C**omplex **I**nstruction **S**et **C**omputer)*
- RISC (**R**educed **I**nstruction **S**et **C**omputer)

There are others (hybrids).

1- CISC Architecture

It was thought that it was more interesting to submit complex instructions to the microprocessor. Thus, rather than encoding a complex operation by several smaller instructions, which would require as much very slow memory access, it seemed preferable to add to the microprocessor's instruction set a complex instruction that would perform this operation.

- Architecture with a large number of instructions.
- The microprocessor must perform complex tasks with a single instruction.
- For a given task, a CISC machine executes a small number of instructions, but each requires a greater number of clock cycles.
- The machine code of these instructions varies from instruction to instruction and therefore requires a complex decoder (microcode).

2- RISC Architecture

Statistical studies have shown that programs generated by compilers are most often satisfied with assignments, additions and multiplications by constants. Thus, 80% of high-level language processing used only 20% of microprocessor instructions. Hence the idea of reducing the instruction set to the most commonly used ones and improving processing speed.

RISC vs CISC Architecture

The choice will depend on the intended applications. Indeed, if we reduce the number of instructions to perform a treatment, we create complex instructions (CISC) that require more cycles to be decoded. And if we decrease the number of cycles per instruction, we create simple instructions (RISC) but then we increase the number of instructions needed to perform the same process.

RISC Architecture	CISC Architecture
1. Simple instructions, taking only one clock cycle	1. Complex instructions, taking several clock cycles
2. Fixed format instructions	2. Instructions in variable format
3. Simple decoder (wired)	3. Complex decoder (microcode)
4. Many registers	4. Few registers
5. Only Upload and Backup instructions have access to memory	5. All instructions are likely to access memory
6. Few addressing modes	6. Lots of addressing modes
7. Complex compiler	7. Simple compiler

Table 1: RISC vs CISC Architecture

3.2.4.8- Core Architecture Enhancements

All microprocessor improvements are aimed at reducing program execution time. The first idea that comes to mind is to simply increase the frequency of the microprocessor's clock. However, the acceleration of frequencies causes an increase in consumption, which leads to a rise in temperature.

For this reason, the work is aimed at lowering the CPI. To do this, there are several ways that mainly affect the internal architecture of the CPU. These techniques include:

3.2.4.8.1- Pipeline Architecture

The execution of an instruction is broken down into a succession of steps and each step corresponds to the use of one of the functions of the microprocessor.

Example of the 4-phase execution of a statement:

Figure 19: Phases of execution of a statement

Classic model:

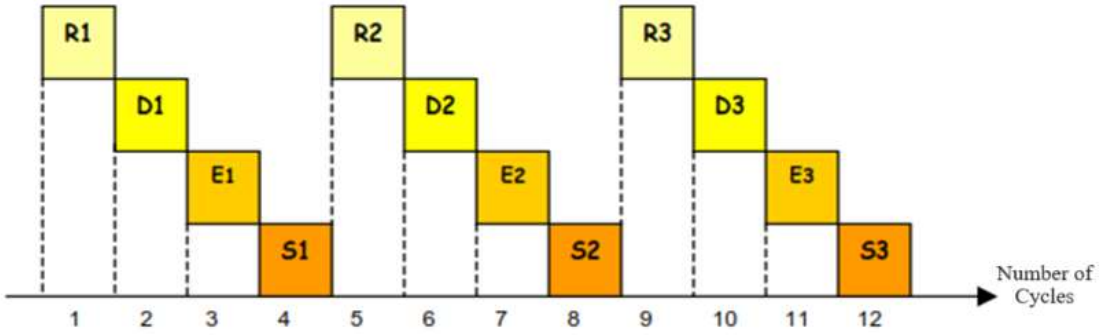


Figure 20: Phases of execution of a statement, classic model

Piped model:

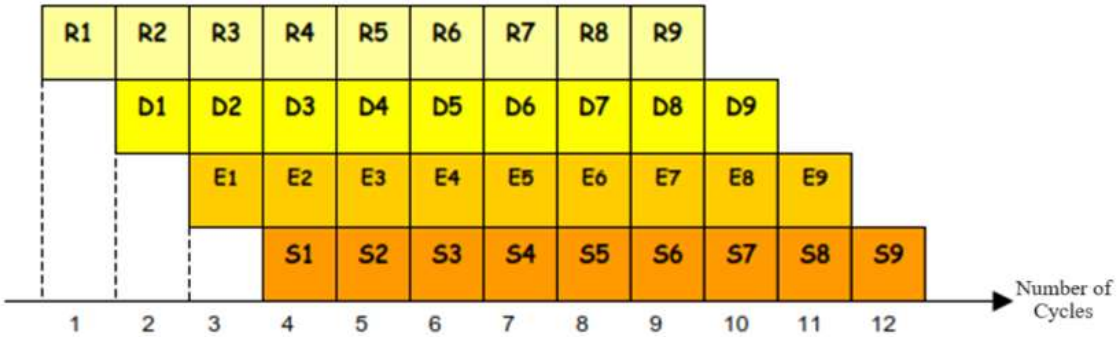


Figure 21: Execution phases of a statement, piped model

Performance gain

The machine starts executing an instruction each cycle. The pipeline is fully occupied from the fourth cycle onwards. The gain obtained therefore depends on the number of stages of the pipeline.

To execute ***n*** instructions, assuming that each instruction executes in ***k*** clock cycles, you must:

- ***n*k*** Clock cycles for sequential execution (classical model).
- ***k*** clock cycles to execute the first instruction then ***n-1*** cycles for the following ***n-1*** instructions if a ***k***-stage pipeline is used

The gain obtained is therefore of:

$$G = \frac{n * k}{k + n - 1}$$

When the number ***n*** of instructions to be executed is large with respect to ***k*, *we can assume that we divide the execution time by* k.**

- AMD's Athlon includes an 11-stage pipeline.
- Intel's Pentium 2, 3, and 4 include a 12, 10 and 20 to 35 stage pipelines, respectively. In the Core series II processors (i3, i5, and i7), there are 14 stages in the processor pipeline.
- Regarding other modern processors:
  - ARM up to 7: 3 stages (still widely used is simpler devices)
  - ARM 8-9 : 5 stages ;
  - ARM 11 : 8 stages ;
  - Cortex A7 : 8-10 stages ;
  - Cortex A8 : 13 stages ;
  - Cortex A15 : 15-25 stages.

Pipeline issues (hazards)

There are several issues (hazards) with setting up a pipeline. In fact, the longer the pipeline, the higher the number of cases where it is not possible to achieve maximum performance. There are three (3) main cases where the performance of a pipelined processor can be degraded. These cases of performance degradation are called **hazards.**

- **Structural hazard** which corresponds to the case where two statements need to use the same CPU resource (dependency conflict).
- **Data hazard** which occurs when one statement produces a result and the next statement uses that result before it can be written to a register.
- **Control hazard** which occurs whenever a branch instruction is executed (the branching prediction mechanism gives a reliability of 90 to 95%).

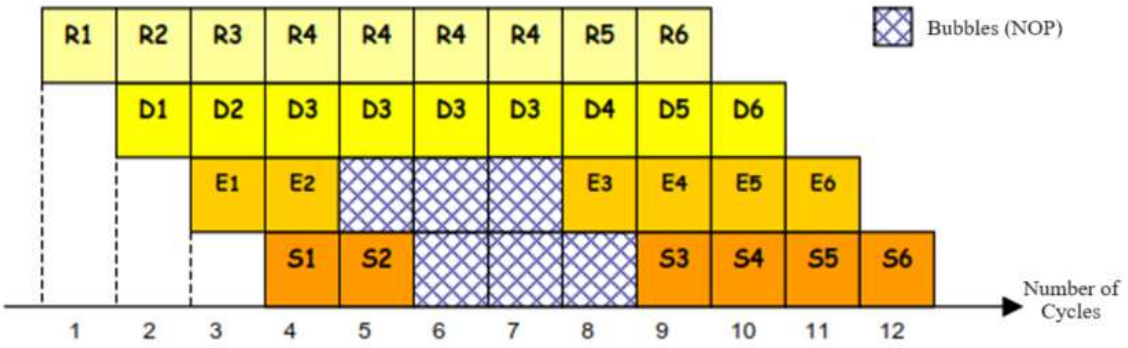


Figure 22: Pipeline hazards

3.2.4.8.2- Cache Memory

There is access latency between memory and the processor because the memory is no longer able to deliver information as quickly as the processor is able to process it.

One of the solutions used to hide this latency is to have a very fast memory between the microprocessor and the memory. This is called a **memory cache**.

This compensates for the low relative speed of the memory by allowing the microprocessor to acquire data at its own speed (**SRAM**, Static RAM, of reduced size).

Its function is to store the most recent or most often used information by the microprocessor.

Now, it is integrated into the microprocessor and even comes on several levels (L1, L2, ...).

Cache Successes: It is called cache success, if the required data or instruction is present in the cache and it is then sent directly to the microprocessor. Otherwise, it is called a cache defect.

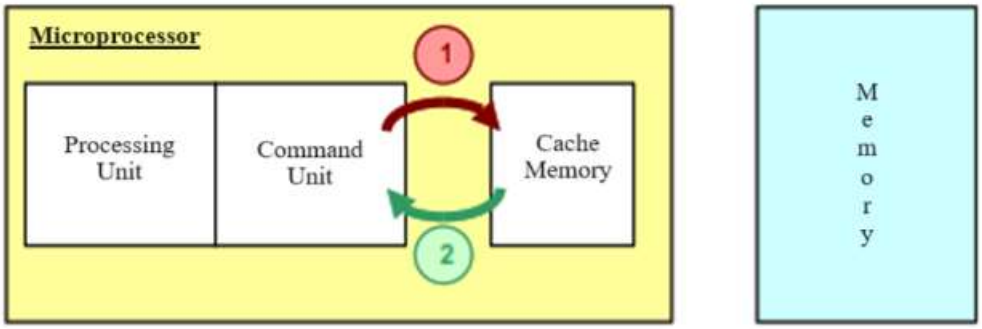


Figure 23: Cache success

Cache fault: 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 sends it back to the microprocessor while storing it in the cache.

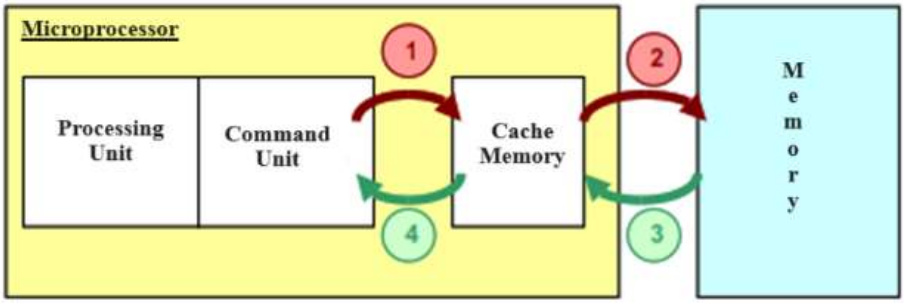


Figure 24: Cache fault

The memory cache provides a performance gain only in the first case. Its performance is therefore entirely linked to its success rate. It is common to experience average success rates in the range of 80 to 90%.

3.2.4.8.3- Superscalar Architecture

Another way to gain performance is to execute multiple statements at the same time. The superscalar approach (implemented in the first Pentium, 1993) consists of equipping the microprocessor with several processing units working in parallel. The instructions are then distributed among the different execution units. It is therefore necessary to be able to support a large flow of instructions and for this to have a powerful cache.

Figure 25: Scalar Architecture

Figure 26: Superscalar Architecture

3.2.4.8.4- Pipeline and Superscalar Architecture

The principle is to execute the instructions in a pipelined way in each of the processing units working in parallel.

PU 1	Search	Decoding	Executing	Save Result
PU 2	Search	Decoding	Executing	Save Result

Figure 27: Superscalar Instruction Architecture

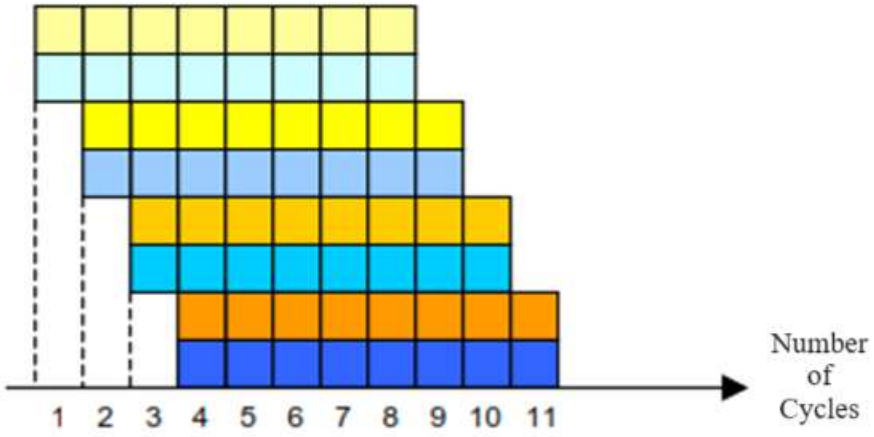


Figure 28: Piped Superscalar Architecture of Instructions

3.3- Special Processors

A- The microcontroller

Minimum system on a single chip. It contains a CPU, RAM, ROM, and I/O ports:

- Also includes specific functions such as programmable meters to perform time measurements, ADCs or DACs to fit into acquisition chains, interfaces for networks, etc...
- Adapted to best meet the needs of embedded applications (domestic appliances, acquisition chain, smart card reader, etc..).
- On the other hand, generally less powerful in terms of speed, treatable data size or addressable memory size than a microprocessor.

B- The Digital Signal Processor, DSP

The DSP (Digital Signal Processor) is optimized to perform digital signal processing (FFT calculation, convolution, digital filtering, etc.).

The fields of application of the D.S.P were originally telecommunications and the military sector. Today, applications have diversified towards multimedia (CD player, MP3, etc.), consumer electronics (digital television, mobile phone, etc.), automation, instrumentation, automotive electronics, etc.

Chapter 4: The Intel 8086 Microprocessor

4.1- Introduction

The objective of this chapter is to understand the architecture and instruction set of the 8086. Then, the following points will be developed:

- **Internal architecture of the 8086**
  - Arithmetic and logical unit (ALU)
  - Control/Command unit
- **Processing and execution of instructions**
  - Assembly language
  - Representation and coding of instructions
- **8086 instructions set**

4.2- Internal Architecture of the intel 8086 Microprocessor

4.2.1- Description of the intel 8086

- Appeared in 1978
- 40-pin DIP (Dual In-line Package) package.
- 16 bits of data
- 20 bits address
- Addresses and data are multiplexed,
- AD0/AD15. The capture of the address is done by the Address Latch Enable (ALE) signal.

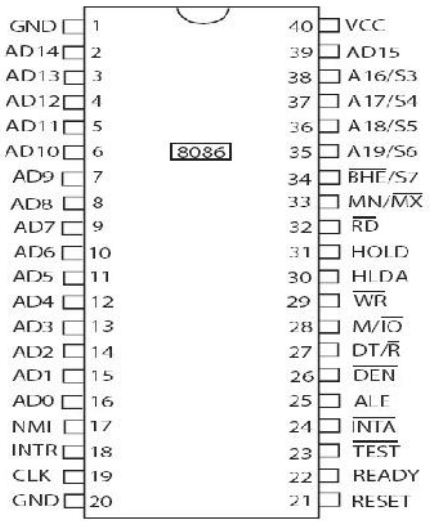


Figure 29: Intel 8086 Chip

4.2.2- The 8086 Microprocessor Registers

- **4 General registers:** AX, BX, CX and DX. Each general register is composed of two registers of 2 bytes (8 bits) (e.g., AX = AH*256 + AL, AH is the high weight register and AL is the low weight register).
- **2 Index registers:** SI (Source Index) and DI (Destination Index)).
- **4 Segment registers:** CS (Code Segment), DS (Data Segment), ES (Extra segment) et SS (Stack Segment).

- **3 pointer registers:** IP (instruction Pointer), SP (Stack Pointer), and BP (Base Pointer).
- **1 Status Register:** FR (Flag Register), register flags are: overflow, carry, auxiliary carry, sign, parity, zero, interrupt, and step-by-step execution.

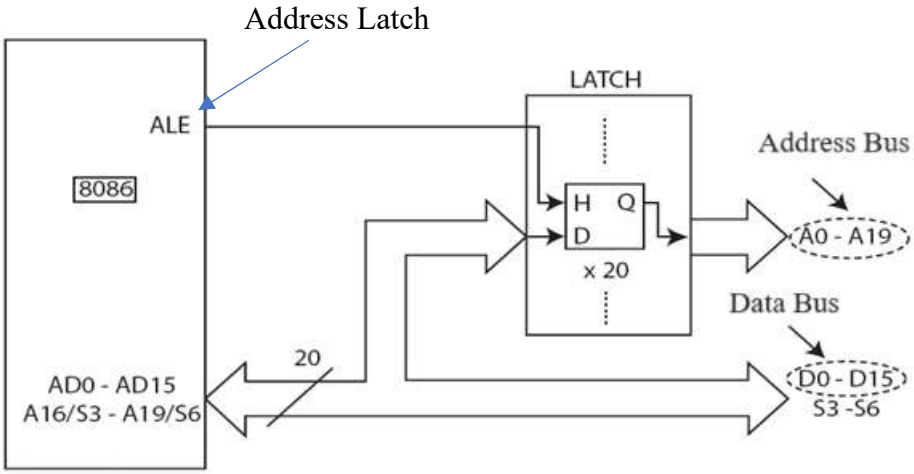


Figure 30: Address / data signal demultiplexing

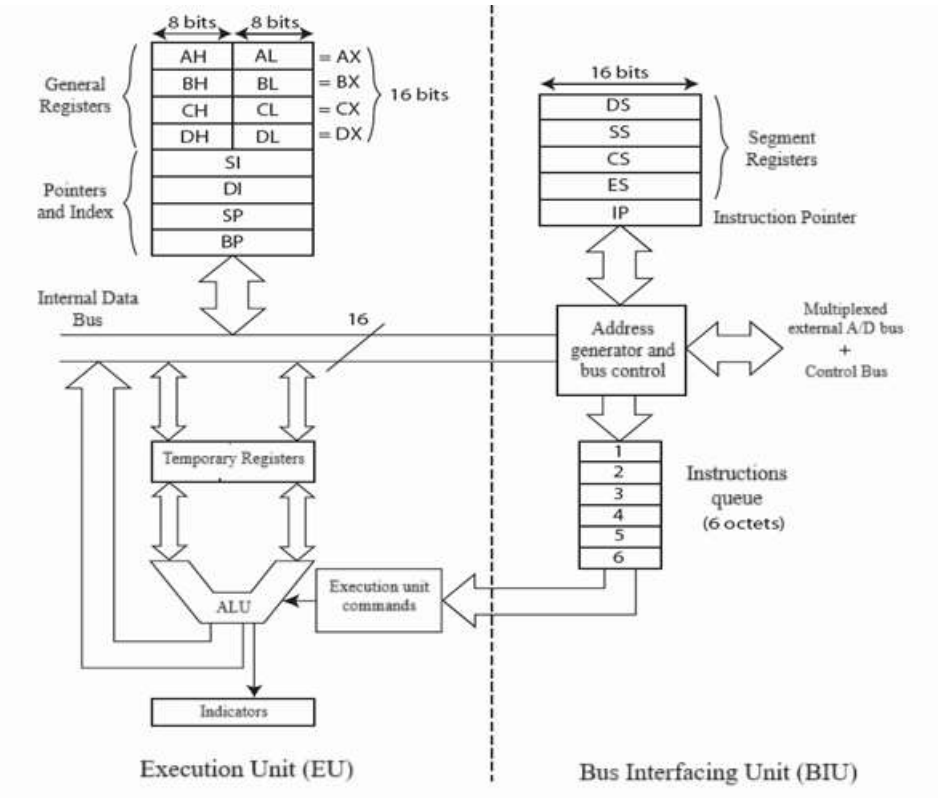


Figure 31: Internal architecture of the 8086

Intel's 8086 microprocessor, like any other processor, consists of running a program written in machine language. The program is a precise sequence of instructions given by the manufacturer.

For ease of reading and work, the manufacturer delivers the product with in addition to the machine language the assembly language.

4.3- Representation and coding of instructions

The instruction is a basic operation that can be decoded and executed by the microprocessor, where each instruction has a given format.

The general format of a statement is:

Mnemonic Operand 1, [Operand 2]

- Mnemonic: is the name of the operation
- Operand 1: is usually a register or memory cell
- Operand 2: is either a register, a memory cell or a value

Example:

```
0100   MOV AX,[DE88]
------ SUB AX,BX
------ INC AX
------ JMP 0240
```

Each instruction is placed in memory in encoded form. This representation is called **machine code**.

The encoding of the instruction is performed by breaking down the instruction into subgroups of bits called fields, specifying the characteristics of the instruction.

The different fields are:

- **Operation code:** code indicating the operation (ADD, MOV, SUB, etc...).
- **Data type:** 1 per byte and 0 per word (2 bytes).
- **Register:** indicates the number of one of the registers used.
- **Addressing mode:** Specifies addressing mode.

Each instruction is encoded in binary and is decoded by the µP when executed. Below are the different codes of the MOV statement:

Machine Code	Assembly language
B8FF00	MOV AX,00FF
A1FF00	MOV AX,[00FF]
89D8	MOV AX,BX
8B07	MOV AX,[BX]

8086 Operation Code Examples

Symbole	Code Op.	Octets	
MOV AX, *valeur*	B8	3	AX ← *valeur*
MOV AX, [*adr*]	A1	3	AX ← contenu de l'adresse *adr.*
MOV [*adr*], AX	A3	3	range AX à l'adresse *adr.*
ADD AX, *valeur*	05	3	AX ← AX + *valeur*
ADD AX, [*adr*]	03 06	4	AX ← AX + contenu de *adr.*
SUB AX, *valeur*	2D	3	AX ← AX - *valeur*
SUB AX, [*adr*]	2B 06	4	AX ← AX - contenu de *adr.*
SHR AX, 1	D1 E8	2	décale AX à droite.

Figure 32: 8086 Operation Code example

4.4 8086 Instructions

4.4.1 Definition

It is the set of instructions that can be executed by the µP. There are several groups of statements:

- Data transfer instructions
- Arithmetic and logic instructions
- Shift and rotation instructions
- The comparison instruction
- Jump and branch instructions
- Instructions relating to the status register

4.4.2- Instruction Description Template

For all instructions that have more than one operand addressing mode, a table showing these modes is given in the following form:

xxx operande1, operande2, ... comment
…
….

In the left-hand column (xxx) is the mnemonic of the instruction. In the next column (operand1), we find the mode of addressing the operands. The right-hand column (operand2), which is absent in some tables, gives some additional information.

The addressing modes are indicated as follows:

- AL, AH, ... a special registry
- Register: one of the 8 or 16 bits register AL, AH, AX, BL, BH, BX, CL, CH, CX, DL, DH, DX, SI or DI.
- Variable: a memory address of a piece of data in the data segment.
- Register/Variable: A register or memory address of a piece of data in the Data segment.
- Constant: one value
- Label: A memory address of an instruction in the code segment.

Most of the statements change the flags in the FLAGS registry. A table shows the effect of the commonly described instruction on these indicators. It has the following form:

O	D	I	T	S	Z	A	P	C

- **Carry Flag (CF):** This indicator is set to 1 when there is a withholding of the result at 8 or 16 bits. It is involved in the operations of addition (carry) and subtraction (borrow) on natural numbers. In particular, it is set by the ADD, SUB, and CMP statements
- **Parity Flag (PF):** If the result of the operation contains an even number of 1, this flag is set to 1, otherwise zero.
- **Auxiliary Flag (AF):** This bit is equal to 1 if we have a carry of the low weight quarter in the higher weight quarter.
- **Zero Flag (ZF):** This flag is set to 1 when the result of an operation is 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.
- **Sign Flag (SF):** SF is set to 1 if the high bit of the result of an addition or subtraction is 1; otherwise, SF=0. SF is useful when working with signed integers, because the high bit gives the sign of the result.
- **Trap Flag (TF):** So that the microprocessor executes the program step by step in debugging mode.
- **Interrupt enable Flag (IF):** To hide interrupts coming from outside, this bit is set to 0, otherwise the microprocessor recognizes the interrupt from outside.

- **Direction Flag (DF):** Auto Increment/Decrement: Used during string statements to auto-increment or auto-decrement the SI and DI.
- **Overflow Flag (OF):** if there is an arithmetic overflow this bit is set to 1. That is to say the result of an operation exceeds the capacity of the operand (register or memory box), otherwise it is 0.

The first line shows the names of the bits of interest in FLAGS. The second line (blank here) indicates the effect of the instruction on a particular bit. We note:

- * : The bit is changed based on the result of executing the statement
- ? : The bit has an undefined value after the statement is executed
- 1 : The bit is set to 1
- 0 : The bit is set to 0

A blank check box indicates that the flag is not changed by the statement.

4.4.3- Transfer instructions

MOV Transfer of Value

These instructions perform data transfers between two memory addresses, two registers, or between a register and memory. This corresponds to the assignment of high-level languages A:=B ($A \leftarrow B$).

MOV register/variable, register
MOV register, register/variable
MOV register, segment register
MOV segment register, register
MOV register/variable, constant

O	D	I	T	S	Z	A	P	C

Segment register, indicates one of the CS, DS, ES, or SS registers.

The MOV instruction transfers data to a registry or memory address. Transferring the value from one segment register to another segment register. There is no addressing mode in the MOV command that allows the value of one segment register to be transferred to another segment register. We only have the ability to transfer the value of a segment register to a data register and the ability to transfer the value from a data register to a segment register. Also, to achieve our goal, we will have to carry out the transfer in two steps, using an intermediate data register:

- Transfer the value from the source segment register to a 16-bit data register.

- Transfer the value from this data register to the destination segment register.

For example, suppose we want to transfer the contents of the DS registry into the ES registry. We can write it in the following form:

mov ax, ds

mov es, ax

XCHG **Exchange of Values**

XCHG register/variable, register XCHG register, register/variable

O	D	I	T	S	Z	A	P	C

The XCHG instruction swaps the contents of two memory or register locations. Suppose the AX register contains the value 88 and the DX register contains the value 99, the execution of the instruction:

xchg ax, dx

gives: AX contains a value of 99 and DX contains a value of 88.

4.4.4- Increment, decrement

Here we see two types of instructions that are very frequently used and which are in fact special cases of addition and subtraction instructions, increment and decrement. Increment means, “add 1”, while decrement means “remove 1”. Note, however, that we often use the terms increment and decrement even if the quantities added or removed are different from 1, generally when the modified variable is a counter.

DEC Decrement

DEC register/variable

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	

DEC subtracts 1 from the contents of the operand, without changing the hold indicator. Let the following assembler instructions be used:

mov al, 10h

dec al

AL then contains the value 0fh. Bits Z, O, P, A and S set to 0.

INC Increment

INC register/variable								
O	D	I	T	S	Z	A	P	C
*				*	*	*	*	

INC adds 1 to the contents of the operand, without changing the carry indicator. Let the following assembler instructions be used:

mov al, 5fh

inc al

AL then contains the value 60h. The Z bit is set to 0 (the result of the increment is not zero), the auxiliary hold indicator bit A is set to 1 (holding pass between bits 3 and 4 during increment), the bit O and bit S are set to 0 (no capacity overflow, the result sign is positive).

4.4.5 Opposite of a number

NEG Negation by complement to 2

NEG register/variable								
O	D	I	T	S	Z	A	P	C
*				*	*	*	*	*

NEG transforms the value of a register or memory operand into its complement to 2. Thus, after executing the instructions:

mov ax, 75h

neg ax

the AX register contains the value ff8bh (complement to 2 of 0075h).

4.4.6 Arithmetic Instructions

As with many processors, the 8086 has +, −, ×, and ÷ instructions that process entire data encoded on a byte or word. To perform operations on more complex data (floating-point numbers, for example), they will need to be programmed.

ADD Addition without carry

ADD register/variable, register
ADD register, register/variable
ADD register/variable, constant

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	*

The ADD statement adds the contents of the source register to the destination register, without carry, i.e.:

destination ← destination + source

The carry is positioned according to the result of the operation.

For example, if the following statements are executed:

mov ax, a9h

add ax, 72h

then the AX register contains the value 11bh, the bit C is set to 1, the auxiliary carry A is set to 0.

If we consider the following two statements;

mov ax, 09h

add ax, 3ah

then the AX register contains the value 43h, the C bit is set to 0, the auxiliary carry A is set to 1.

SUB Subtraction without carry

SUB register/variable, register
SUB register, register/variable
SUB regiser/variable, constante

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	*

The SUB statement subtracts the contents of the source register from the destination register, without carry, i.e.;

destination ← destination – source

The carry is positioned according to the result of the operation.

For example, if the following statements are executed;

mov ax, 39h
sub ax, 18h

the AX register then contains the value 21h. The Z, S, and C bits of the FLAGS register are set to 0 because the result is not zero, its sign is positive, and no carry are generated.
If we consider the following two statements;

mov ax, 26h
sub ax, 59h

the AX register then contains the value ffcdh. The Z bit is set to zero. The C, A, and S bits are set to 1.

IMUL

MUL Multiplications in assembler

IMUL register/variable

O	D	I	T	S	Z	A	P	C
*				?	?	?	?	*

Both IMUL and MUL statements perform multiplications. The IMUL statement multiplies signed operands. The MUL statement multiplies unsigned operands.
The carry (C) and overflow (O) indicators are set to one if the result cannot be stored in the destination operand.

Adding or subtracting data encoded at n bits gives a result of at most n + 1 bits. The extra bit is the retainer and is stored in the C bit of flags. On the other hand, multiplying two data of n bits gives a result of 2n bits.

- In their first form, which takes 8-bit operand data (i.e., the result is 16 bits), the MUL and IMUL statements are the product of the value contained in the AL register with the value of the provided operand. The result is placed in the AX register.
- In their second form, which takes a 16-bit operand data (so the result is 32 bits), the MUL and IMUL statements take the product of the value contained in the AX register with the value of the provided operand. The result is placed in the DX and AX register pair. DX contains the high weight of the result, AX the low weight.

Example of 8-bit multiplication: Let's look at the following instructions:

mov al, 4h
mov ah, 25h
imul ah

at the end of the execution of these 3 instructions, the AH register contains the value 94h, which is the product of 4h by 25h. To understand the difference between IMUL and MUL statements, let's look at the following examples:

```
mov  bx, 543h
mov  ax, 3257h
imul bx
```

at the end of the execution of these 3 instructions, AX contains the value dfc5h and DX the value 108h, i.e. the hexadecimal value 108dfc5h, the product of 543h by 3257h. Since both data are positive, the result is the same whether we use the IMUL statement or the MUL statement.

Now let's consider the sequence of statements:

```
mov  bx, -543h
mov  ax, 3257h
imul bx
```

at the end of their execution, AX contains the value 203bh and DX contains the value fef7h, which is the hexadecimal value fef7203bh, in decimal -17358789. If we replace IMUL with MUL, the result is meaningless (314e203bh, in decimal 827203643, positive value!).

IDIV

DIV Divisions in assembler

IDIV register/variable

O	D	I	T	S	Z	A	P	C
?				?	?	?	?	?

Both DIV and IDIV carry out the operations of dividing and calculating remainder. DIV performs it on unsigned data, IDIV on signed data.

In all cases, the dividend is implicit. The divisor is provided as an operand. The result consists of the quotient and the rest of the division. The remainder is always less than the divisor. You can choose between:

- The division of a 16-bit piece of data stored in AX by an 8-bit piece of data that provides a quotient in AL and a remainder in AH over 8 bits.
- The division of a 32-bit piece of data stored in the DX (high-weight) and AX (low-weight) register pair by a 16-bit piece of data that provides a quotient in AX and a remainder in DX on 16 bits.

Let the following few assembler lines:

```
mov  ax, 65h
mov  dx, 6h
div  dl
```

After they are executed, the AX register contains the value 0510h, quotient of 10h in AL and 05h the rest of the division in AH.

For both statements, if the divisor of the division is zero, a Division by zero message will be displayed automatically.

In the case of an IDIV-signed division, the remainder has the same sign as the dividend and its absolute value is always less than the divisor.

To understand how these two statements work, consider the following example:

```
mov  ax, 3257h
mov  bx, 543h
div    bx
```

At the end of executing this sequence of instructions, the register AX will contain the value 9h which is the quotient of 2372h by 435h, and the register DX will be 2fch which is the remainder of the division. If we replace DIV with IDIV, the result is unchanged since both the divisor and the dividend are positive.

If we now consider the sequence:

```
mov  ax, 3257h
mov  bx, -543h
idiv   bx
```

we will then have AX which contains the value ffff7h (evening 65527 in decimal) and DX the value 2fch (i.e. 764 in decimal). If we replace IDIV with DIV, the result is meaningless.

4.4.7 Boolean and logical instructions

These instructions, which are available on all processors, work on bit-level data (and not numeric values like the instructions seen so far).

AND Ligical AND

AND register/variable, register
AND register, register/variable
AND register/variable, constante

O	D	I	T	S	Z	A	P	C
0				*	*	?	*	0

AND performs a bit-by-bit logic between the source operand and the destination operand. The result is stored in the destination operand. Consider the following two lines:

mov al, 56h

and al, 2ch

The AL register then contains the 14h value obtained as follows:

56h	0101	0110
∧ 2ch	0010	1100
14h	0001	0100

The S and Z bits are set to zero and the P bit is set to 1.

OR Logical OR

OR register/variable, register
OR register, register/variable
OR register/variable, constante

O	D	I	T	S	Z	A	P	C
0				*	*	?	*	0

OR performs a bit-by-bit logical ou-logic between the source operand and the destination operand. The result is stored in the destination operand. Consider the following two lines:

mov al, 56h

or al, 2ch

The AL register then contains the resulting value 5eh as follows:

56h	0101	0110
∨ 5ch	0101	1100
5eh	0101	1110

The S and Z bits are set to zero and the P bit is set to 1.

XOR Logical Exclusive-OR

XOR register/variable, register
XOR register, register/variable
XOR register/variable, constante

O	D	I	T	S	Z	A	P	C
0				*	*	?	*	0

XOR performs a bit-a-bit exclusive or--bit between the source operand and the destination operand. The result is stored in the destination operand. Consider the following two lines:

mov al, 73h

and al, 6bh

The AL register then contains the 18h value obtained in the following way:

	73h	0111	0011
⊕	6bh	0110	1011
	18h	0001	1000

The S and Z bits are set to zero and the P bit is set to 1.

NOT Logical Negation

NOT register/variable

O	D	I	T	S	Z	A	P	C

NOT transforms the value of a register or operand into its bit-by-bit logical complement. Consider the following two lines:

mov al, 59h

not al

The AL register then contains the value a6h obtained as follows:

Not 59h	0101 1001
a6h	1010 0110

The S and P bits are set to 1, the Z bit to 0.

4.4.8 Assembler tests

In assembler, there are no tests like in high-level languages such as Pascal or C. However, it is of course possible to carry out tests. This is done using the bits of the FLAGS register as a test condition and a conditional branching instruction (jump if some bits of the FLAGS register are 0 or 1) to trigger the then or otherwise part of the test.

General principle

The bits of the FLAGS register are positioned by the instructions we have already seen (arithmetic instructions, logical instructions, ...). They are also positioned by instructions specifically designed for testing and which have no other effect than to position the bits of FLAGS according to certain conditions on their operands. These instructions are CMP and TEST.

Once the flags are set, a so-called conditional jump instruction tests a bit or a combination of bits of FLAGS and, depending on the result:

- Performs a sequence break (a jump) to a specific location in the code where execution continues normally.

Continue in sequence if the test does not give a positive result.

We present the CMP statement, the conditional and unconditional jump statements and then present the coding of the tests on examples.

Comparison Instruction

CMP Comparison

CMP register/variable, register
CMP register, register/variable
CMP register/variable, constante

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	*

This is the most commonly used statement for positioning flags before performing a conditional jump statement.

CMP allows you to compare two values. To do this, CMP subtracts the second operand from the first, without modifying the destination operand, but by positioning the indicators according to the result. So, if the subtraction result is zero, so the indicator Z has been set to 1, it means that the two values being compared are equal. Using reasoning of the same kind, we can find out whether the two values are different, strictly ordered or not.

Upon completion of the execution of the following two statements:

mov al, 23h

cmp al, 34h

The AL register is not modified by the execution of the CMP statement and still contains the previously assigned value at 23h. The carry indicator C is set to 1, indicating that the second operand of the CMP statement is greater than the value of the first operand. Since the zero Z indicator is set to 0, this indicates that the two data are different (otherwise, subtracting a number from itself gives 0, so the Z bit is set to 1). The S-sign bit is also set to 1 because 23h – 34h is a negative number.

Jxx Conditional Jump Instructions

All conditional jump statements take the same type of operand:

JA	Label	Jump if above (C =0 et Z =0)
JAE	Label	jump if above than or equal (C =0)
JB	Label	Jump if below (C =1)
JBE	Label	Jump if below than or equal (C =1 ou Z
JC	Label	Jump if carry (C =1)
JCXZ	Label	jump if CX is 0
JE	Label	Jump if equal (Z =1)
JG	Label	Jump if greater (Z =0 ou S =0)
JGE	Label	jump if greater than or equal (S =0)
JL	Label	Jump if lower (S =0)
JLE	Label	Jump if less than or equal (Z =1 ou S =0)
JNC	Label	Jump if no carry (C =0)
JNE	Label	jump if not equal (Z =0)
JNO	Label	Jump if no overflow (O =0)
JNP	Label	Jump if no parity (P =0)
JNS	Label	jump if no sign (S =0)
JO	Label	Jump if overflow (O =1)
JP	Label	Jump if parity (P =1)
JS	Label	Jump If Sign (S =1)

Figure 33: 8086 Conditional Jump Instructions

All of these statements work as follows: When the condition is true, a jump is made to the statement at the label specified in operand. Otherwise, the execution sequence is followed.

Note that lower-order tests (lower, higher, etc.) can be understood in the following way. Suppose we compare two pieces of data by a CMP statement and then execute a JG statement,

i.e., "jump if greater". There will then be a jump if the value of the second operand of the CMP statement is greater than the value of the first operand.

Remark: The label referenced in the conditional break statement should not be too far from the jump statement. Otherwise, an assembly error is triggered and the message:

Relative jump out of range by xxx bytes

The "LOOP" Looping Instruction

LOOP Lab								
O	D	I	T	S	Z	A	P	C

The "loop" statement does the same function as the "for i:=N downto 0 do" statement in algorithmic language. It executes all the instructions between "loop" and the label (the address) to which the "loop" statement refers. The N value will be stored in the CX register and it is decremented after each execution to "0". It decrements the content of CX by 1.

- If CX is non-zero then IP = IP + displacement
- If CX = 0, the following statement is executed.

Example:

```
        MOV   AX, 05
        MOV   CX, 05
Label : INC   AX
        ADD   AX, CX      SI CX < > 0
        LOOP  Label
SI CX =0
        MOV   BX, AX
```

The execution of the MOV BX, AX statement will be done after the loop has been executed 5 times.

JMP Unconditional Jump

JMP Label								
O	D	I	T	S	Z	A	P	C

The JMP statement makes an unconditional jump to the specified label. Unlike a conditional jump, the jump is always made, with the FLAGS registry not interfering with this operation.

CALL, concept of procedure (function)

The notion of an assembly procedure corresponds to that of a function in C language, or of a subroutine in other languages.

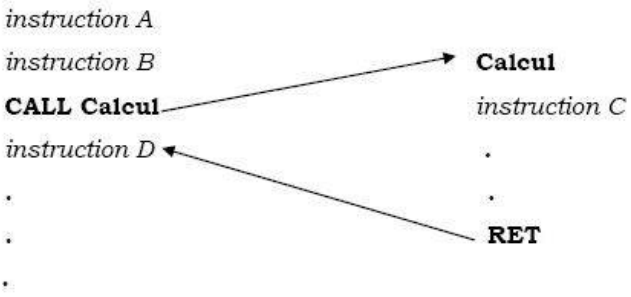


Figure 34: Calling Procedure

The procedure is called *calculus*. After statement *B*, the processor moves to instruction *C* in the procedure, then continues until it encounters RET and returns to instruction *D*.

A procedure is a sequence of instructions that perform a specific action, which are grouped together for convenience and to avoid having to write them repeatedly in the program.

Procedures are identified by the address of their first statement, which is associated with an assembler label.

The execution of a procedure is triggered by a calling program. One procedure can itself call for another procedure, and so on.

CALL and RET Instructions

The call to a procedure is made by the CALL statement.

CALL Address_beginning_procedure

The address is 16 bits, so the procedure is in the same instruction segment. CALL is a new unconditional branching statement. The end of a procedure is marked by the RET statement.

RET :

RET doesn't take arguments, the processor switches to the instruction placed immediately after the CALL.

RET is also a branching statement: the IP register is modified to return to the value it had before the call by CALL. How does the processor find this value? The problem is complicated by the fact that one can have any number of nested calls, as shown in the following figure (36):

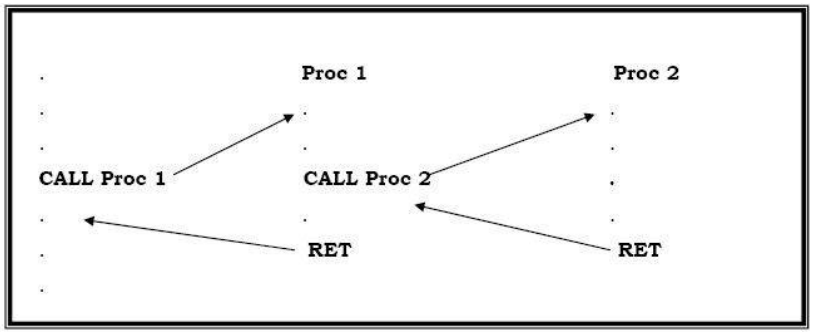


Figure 35: Calling Nested Procedures

The return address, used by RET, is actually saved on the stack by the CALL statement. When the processor executes the RET instruction, it unstacks the address on the stack (such as POP), and stores it in IP.

The CALL statement therefore performs the following operations:

- Stack the value of IP. At this point, IP points to the instruction that follows the CALL.
- Place in IP the address of the first statement of the procedure (given as an argument).

And the RET statement:

- Unstack a value and store it in IP.

4.4.9 The Stack

The Stack is an area of memory that allows you to quickly store and retrieve values for:

- Place local variables in a subroutine,
- Save the return address (done by CALL, INT statements),
- Pass arguments to a subroutine.

A Stack works like a stack of real objects, in LIFO (Last In First Out) mode:

- It is possible to add a value to the top of the stack (stack),
- It is possible to remove the value at the top of the stack (unstack).

Instruction PUSH:

It allows the CPU registers to be stacked on top of the stack.

```
PUSH  Source
```

Instruction POP:

It allows you to unstack the CPU registers on the top of the stack

```
POP     Destination
```

Example:

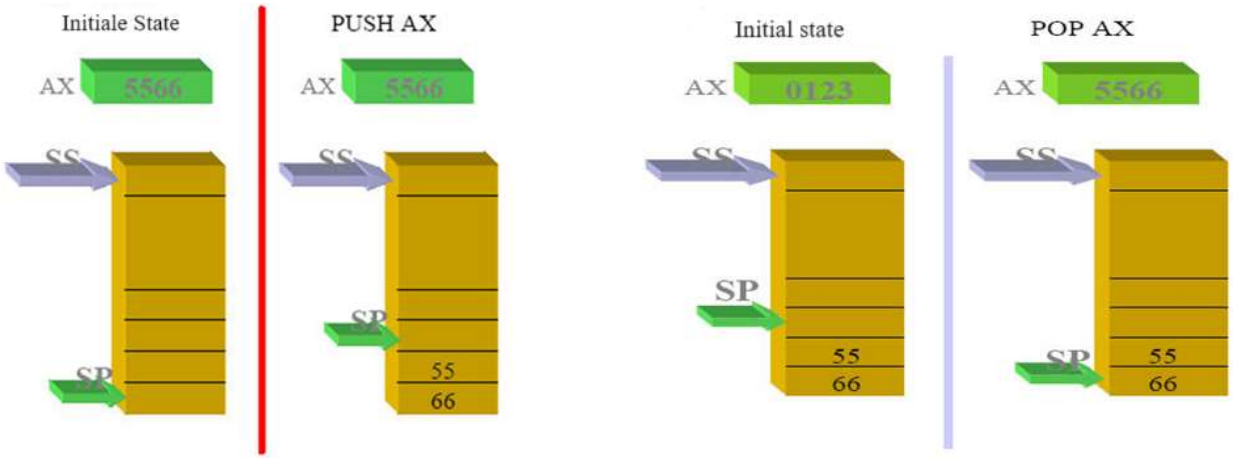


Figure 36: Operating principle of PUSH and POP instruction

4.4.10 Input-Output Instructions:

IN / OUT:

It allows you to retrieve data from a port (i.e., from the edge) or to return data to a port. In both cases if it is a question of sending or receiving a byte we use the AL accumulator, if it is a question of sending or receiving a word, we use the AX accumulator.

Syntax:

```
IN      ACCUMULATOR, DX
OUT     DX,     ACCUMULATOR
```

DX: Contains the address of the port.

ACCUMULATOR: contains the data (to be received or send).

4.4.11 Shift and Rotate Instructions

Here we describe classical operations, offsets, and rotations. They are commonly encountered because their use greatly simplifies certain treatments.

RCL ROL RCR ROR Rotations in assembler

The various rotation instructions for the 8086 are summarized in the following figures:

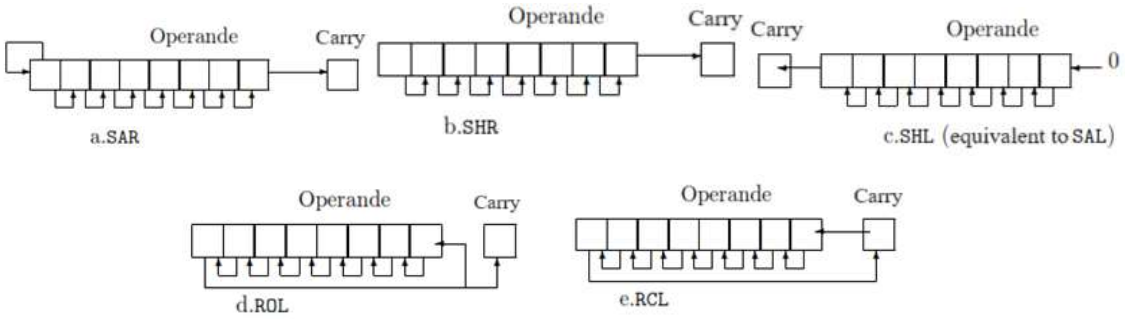


Figure 37: Shifts and rotations in assembler

Rotations are frequently used binary logical operations. They consider an operand (byte or word) as a torus whose bits they shift. When offset, a bit overflows to one side, left or right, depending on the direction of rotation. Depending on the case, a few details differ:

RCL the high-weight bit is set in the carry indicator C, the value of this indicator being previously set in the low-weight bit (figure 38.e)

ROL the high bit is set in the carry indicator C and in the low weight bit of the operand. The old value of the holding indicator is not used (Figure 38.d).

The operands of the RCL, RCR, ROL, and ROR statements, being the same, we present only one, the RCL instruction.

RCL	register/variable, 1
RCL	Register/variable, CL

O	D	I	T	S	Z	A	P	C

- **RCL** rotates to the left of the specified destination operand, 1 or CL times, taking into account the contents of the carry indicator. The high bit of the destination operand is set in the carry. The contents of the carry are put into the low-order bit of the destination operand.
- **RCR** Rotates to the right of the specified destination operand, 1 or CL times, taking into account the contents of the carry indicator. The low-order bit of the destination operand is put into the carry. The contents of the carry are put into the high bit of the destination operand.

- **ROL** rotates to the left of the specified destination operand, 1 or CL times, regardless of the contents of the carry indicator. The high bit is put into the carry indicator when rotating as well as the low bit of the operand.
- **ROR** Rotates to the right of the specified destination operand, 1 or CL times, regardless of the contents of the carry indicator. The low weight bit is put in the carry indicator during rotation as well as in the high bit of the operand.

Consider the following lines:

```
Mov   al, 16h
Mov   cl, 3
xxx   al, cl
```

where "xxx" is a rotation statement. Depending on the choice of this instruction, we obtain the following results:

xxx	rcl	rcr	rol	ror
al	b0h	85h	b0h	85h
C	0	1	0	1

SAL SHL SAR SHR Shifts in assembler

A shift operation simply involves shifting all the bits of a piece of data. Unlike rotations which consider data (a byte or a word) as a torus, a shift considers the data as a queue; thus, the bit that “overflows” the carry is lost.

The operands of the SAL, SAR, SHL and SHR instructions being the same, we only present one instruction, SAL.

```
SAL  register/variable, 1
SAL  register/variable, CL
```

O	D	I	T	S	Z	A	P	C
*				*	*	?	*	*

SAL and SHL are synonyms and can be used interchangeably. SAL shifts left, saving the high order bit in the carry flag and putting a 0 in the low order bit (Figure 38.c).

SHR shifts to the right. The least significant bit is put into carry. A 0 is put in the most significant bit of the data (figure 38.b).

SAR shifts right, saving the low bit in the carry flag.

Furthermore (and this is the difference with the previous instruction), the most significant bit is kept (figure 38.a). The most significant bit of the initial data is therefore duplicated.

Consider the following three lines:

```
mov   al, 16h
mov   cl, 3
sal    al, cl
```

The AL register then contains the value b0h. Indeed, 16h is 00010110 written in binary. If we shift this value three positions to the left, we get 10110000, which is b0h The C bit is set to 0. Consider the following three lines:

```
mov   al, 36h
mov   cl, 3
sar   al, cl
```

The AL register then contains the value 05h. The C bit is set to 1. The difference between SAR and SAL instructions only appears if the high bit of the data is 1. The following table gives the effect of the different shifts for two values of the data (put in the AL register).

al	f0	70
sar al, 1	f8	38
shr al, 1	78	38

It should be noted that for a number, a shift of one position to the left corresponds to a multiplication of this number by 2 and that a shift of one position to the right corresponds to an integer division by 2. Generalizing, a shifting l positions to the left corresponds to a multiplication by 2 and shifting l positions to the right corresponds to a division by 2. It should be noted, and this is what justifies the existence of the two instructions SAR and SHR and their subtle difference, that SAR performs division on a number in signed representation while SHR performs division on a number in unsigned representation.

Of course, shift operations being known as binary logic operations and not arithmetic operations, care should be taken when interpreting the overflow or carry indicators that will be made at the end of a shift instruction.

Chapter 5: Memories

5.1- Introduction

A memory is a circuit used to record, store and retrieve information (instructions and variables). It is this ability to memorize that explains the versatility of digital systems and their adaptability to many situations.

The information can be written or read.

- Written = saving information in memory,
- Read = retrieval of previously saved information.

5.2- Organizing a memory

A storage unit can be represented as a storage cabinet made up of different drawers. Each drawer then represents a memory box (slot) which can contain a single element: data. The number of memory slots can be very high, it is then necessary to be able to identify them by a number. This number is called address. Each piece of data then becomes accessible thanks to its address.

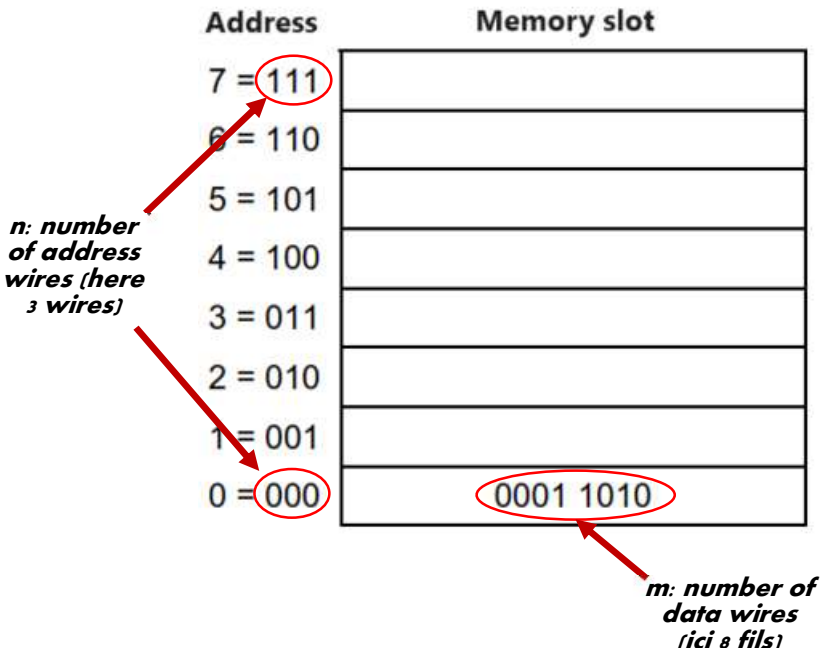


Figure 38: Typical Representation of a Memory

With an '*n*' bit address it is possible to reference at most 2n memory slots (cells) = addressable space. Each cell is filled with a data word (its length *'m' is always a power of 2, 8, 16, 32, 64). The number of address wires in a memory enclosure therefore defines the number of memory slots in the enclosure. The number 'm' of data wires defines the size of data that can be saved in each memory slot.*

In addition to the address bus and data bus, a memory enclosure includes:

- A command entry that allows you to define the type of action you perform with the memory (read/write), $R/\overline{W}$.
- A selector input that allows the inputs/outputs of the enclosure to be set to high impedance, $\overline{CS}$.

We can therefore schematize a memory circuit by the following figure:

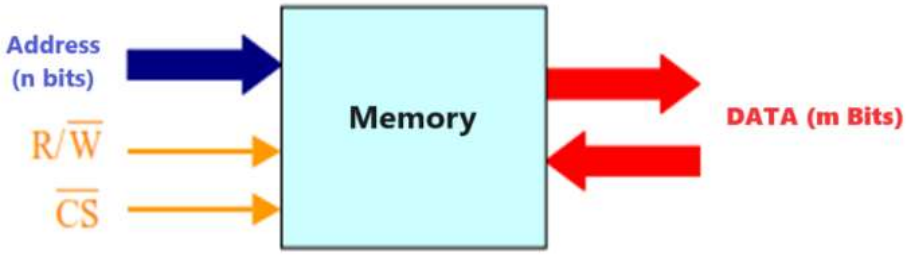


Figure 39: Elements of a Memory Circuit

A memory read or write operation always follows the same cycle:

1- Address Selection
2- Choosing the operation to be performed ($R/\overline{W}$)
3- Memory Selection ($\overline{CS} = 0$)
4- Reading or writing data

Remark:

Data inputs and outputs are very often grouped on **bidirectional terminals**.

5.3- Characteristics of a memory

5.3.1- Capacity

This is the total number of bits in memory. It's also often expressed in bytes, so that's the number of slots. Memory capacity (size) is the number of slots, usually expressed in kilobytes or megabytes, or even more (Terabytes).

Remark: - The kilo of computer is 1024 and not 1000 ($2^{10} = 1024 \approx 1000$).

Here are the most commonly used multiples:

- 1 K (Kilo), 2^{10} = 1024
- 1 M (Méga), 2^{20} = 1048 576
- 1 G (Giga), 2^{30} = 1 073 741 824
- 1 T (Téra), 2^{40} = 1 099 511 627 776

- 1 P (Péta), 2^{50} = 11 258 999 906 842 624

5.3.2- The format of the data

This is the number of bits that can be memorized per memory slot (4, 8, 16, 32, 64...). We also say that it is the width of the memorized word.

5.3.3- Access time

This is the time that elapses between the moment when a read/write operation in memory was launched and the moment when the first information is available on the data bus.

5.3.4- Le temps de cycle

It represents the minimum interval between two successive read or write requests.

5.3.5- Throughput

C'est le nombre maximum d'informations lues ou écrites par seconde.

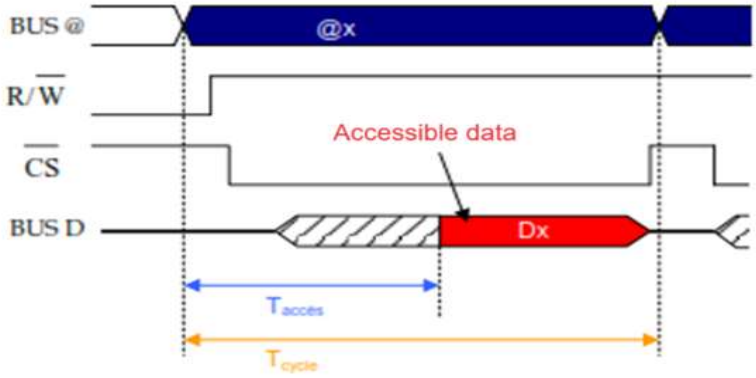


Figure 40: Chronogram of a reading cycle

5.3.6- Volatility

It characterizes the permanence of information in memory. The stored information is volatile if it is likely to be altered by a power supply fault and non-volatile otherwise.

5.3.7- Modes of access

5.3.7.1- Random or direct access

The memories used to make **the main memory** of a microprocessor system are solid-state memories. In this type of memory, you can directly access any information whose address you know and the time it takes to get that information does not depend on the address. It will be said that access to such memory is **random** or **direct**.

5.3.7.2- Sequential Access

To access information on magnetic tape, you have to unroll the tape, locating all the recordings until you find the one you want. Access to information is said to be **sequential.** The access time varies depending on the position of the information sought.

5.3.7.3- Semi-sequential access

Access can still be **semi-sequential** : a combination of direct and sequential access. For a magnetic disk (hard disk) for example, access to the track is direct, then access to the sector is sequential.

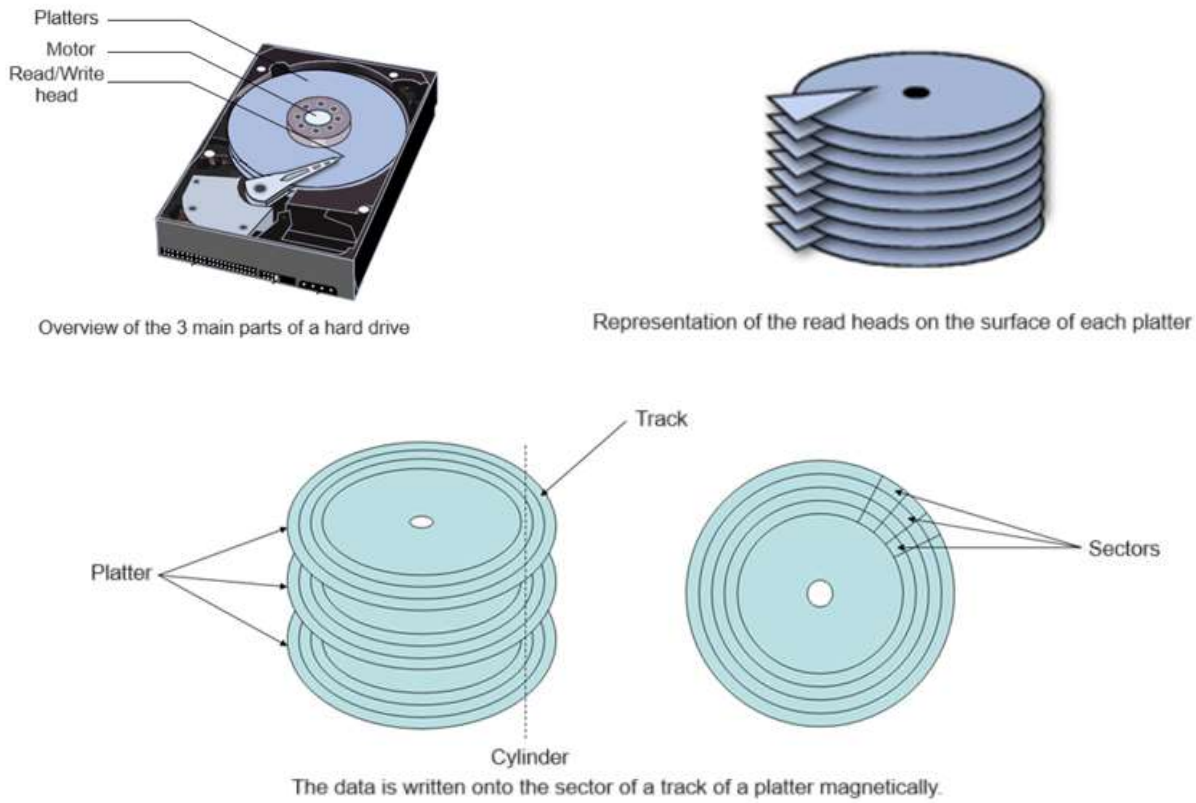


Figure 41: Composition and operation of a magnetic hard drive

5.4- Different types of memory

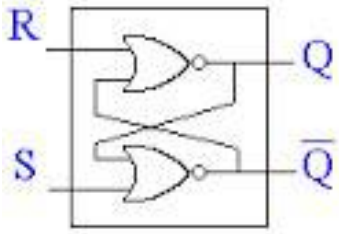
5.4.1- Random Access Memory (RAM)

RAM (**R**andom **A**ccess **M**emory) is RAM used for temporary data storage. It must have a very short cycle time so as not to slow down the microprocessor. They are, in general, volatile. There are two main families of RAM memory:

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

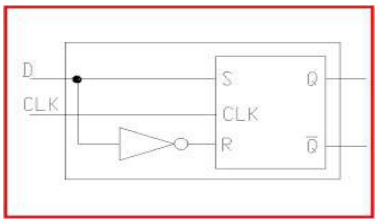
5.4.1.1- Static RAM (SRAM)

In static RAM (SRAM), the memory bit (point) is made up of a flip-flop.



R	S	Q	$\overline{Q}$
0	0	Previous state	
0	1	1	0
1	0	0	1
1	1	Banned state	

Figure 42: RS flip-flop and its truth table



D	H	Q	$\overline{Q}$
φ	0	Previous state	
φ	↑	Previous state	
0	↑	0	1
1	↑	1	0

Figure 43: D flip-flop and its truth table

Each flip-flop contains between 2 to 6 transistors.

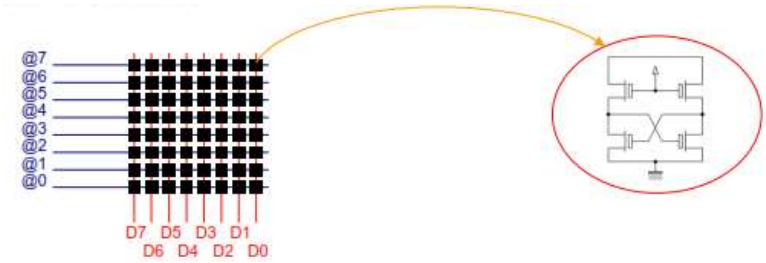


Figure 44: SRAM Memory Point Equivalent in Transistors

5.4.1.2- Dynamic RAM (DRAM)

In dynamic RAM (DRAM), information is stored in the form of an electrical charge stored in a capacitor (grid capacitor substrate of a MOS transistor). 1 to 2 transistors.

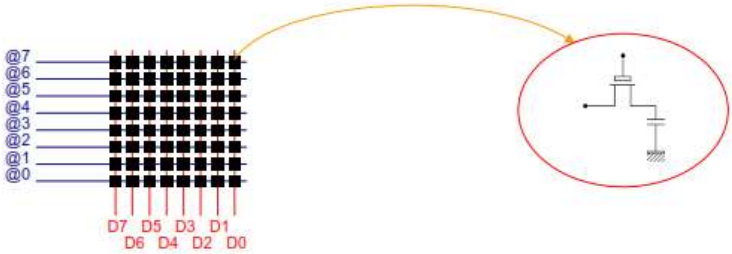


Figure 45: DRAM Memory Point Equivalent in Transistors

5.4.1.2.1- Advantages of Dynamic RAM

Dynamic memory has the following advantages:

- Higher integration density, as a memory point requires about **four times fewer** transistors than in static memory.
- Very low consumption.

5.4.1.2.2- Disadvantages of Dynamic RAM

- Capacitor discharge due to leakage currents.
- Information is lost if it is not regenerated periodically.
- Dynamic RAMs therefore need to be refreshed regularly to maintain memorization
- This **much-needed refresh** has several consequences:
  - It complicates the management of dynamic memories (periodic refresh).
  - The duration of these actions increases the time it takes to access information.
- Reading information is destructive. This is done by discharging the capacity of the memory point when it is loaded. So every reading must be followed by a rewrite.

5.4.2- Criteria for choosing between SRAM and DRAM

Dynamic Memory (DRAM), which provides

- Greater density of information
- lower cost per bit,

are used for **central memory (main memory)**.

Static memory (SRAM),

- faster,

are used when **the speed factor is critical**, especially for small memory sizes such as **caches and registers**.

5.4.3- Read Only Memories

For some applications, it is necessary to be able to retain information permanently even when the power supply is interrupted. In this case, read-only memories are used (**ROM**: **R**ead **O**nly **M**emory).

These memories are:

- Non-volatile.
- Can only be read.
- Storing data in memory is still possible but is called **programming**. Depending on the type of ROM, the programming method will change.

There are several types of read-only memories:

- ROM
- PROM
- EPROM
- EEPROM
- FLASH EPROM.

5.4.3.1- The ROM

It is programmed by the manufacturer and its contents can no longer be modified or deleted by the user.

This memory is composed of a matrix whose programming is carried out by connecting the rows to the columns by diodes. The address is used to select a row in the matrix and the data is then received on the columns. The number of columns that set the size of memory words.

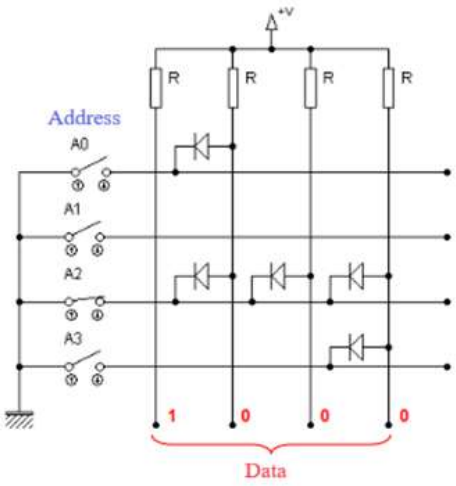


Figure 46: Principle of manufacturing a ROM

For programming, the user must provide the manufacturer with a *mask* showing the locations of the diodes in the array.

Advantages:

- High Density
- Non volatile
- Fast Memory

Disadvantages:

- Unable to write
- Impossible to modify (any mistake is fatal).

- Manufacturing time
- Requirement of large quantities due to the high cost of mask production and manufacturing process.

5.4.3.2- The PROM

It is a ROM that can be programmed only once by the user (**Programmable ROM**). Programming is carried out using a specific programmer.

The diode links in the ROM are replaced with fuses that can be destroyed or junctions that can be shorted.

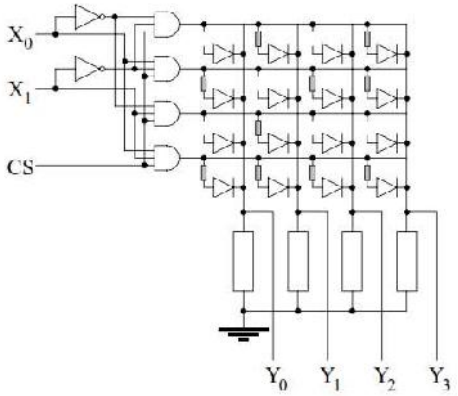


Figure 47: Principle of manufacturing a PROM

Fused PROMs come with all rows connected to the columns (0 at each memory point). The programming process consists of programming the locations of the "1" by generating pulses of currents via the programmer. The fuses located at the selected memory points are therefore destroyed.

Junction PROMs, the principle is identical, except that the rows and columns are disconnected (1 at each memory point). The programming process therefore consists of programming the locations of the “0” by generating current pulses via the programmer; the junctions located at the selected memory points finding themselves short-circuited by an avalanche effect.

Advantages:

- same as ROM
- Breakdown in minutes
- Relatively low cost

Disadvantages:

- Impossible to modify (any mistake is fatal).

5.4.3.3- EPROM or UV-EPROM

To facilitate the development of a program or simply to allow a programming error, it is interesting to be able to reprogram a PROM.

The solution is in the **EPROM** *(**E**rasable **P**rogrammable **R**OM)* is a PROM that can be erased.

In an EPROM, the memory point is made from a FAMOS (**Floating gate Avalanche injection Metal Oxide Silicon)** transistor.

This MOS transistor was introduced by Intel in 1971 and has the particularity of having a floating gate.

To erase this type of memory, it is exposed to about twenty minutes of ultraviolet radiation (hence the name UV-EPROM) to cancel the charge stored in the floating grid.

Advantages:

- Reprogrammable
- Non volatile

Disadvantages:

- Unable to select a single cell to delete
- Unable to erase in-situ memory.
- Writing is much slower than on RAM (Approx. 1000x)

5.4.3.4- The EEPROM

EEPROM (**Electrically EPROM**) is a programmable and electrically erasable memory. It thus addresses the main disadvantage of EPROM and can be programmed in situ.

Advantages

- Behavior of non-volatile RAM.
- Word-by-word programming and erasure possible.

Disadvantages

- Very slow for RAM use.
- Cost of realization.

5.4.3.5- Flash Memory

Flash memory is a type of **EEPROM** that allows multiple memory spaces to be changed in a single operation. This makes flash memory faster when the system needs to write to multiple places at the same time. It was invented by **Prof. Fujio Masuoka** in **1980** (a **Toshiba employee**).

There are two different technologies that differ in the organization of their memory networks which is related to the logic gates used (**NOR** and **NAND).** So, we're talking about:

- Architecture NOR
- Architecture NAND

5.4.3.5.1- Flash NOR

It was the first to be commercially, developed by Intel in **1988**. The **NOR** architecture proposes an assembly of the elementary memory cells **in parallel** with the selection lines as in a classic EEPROM.

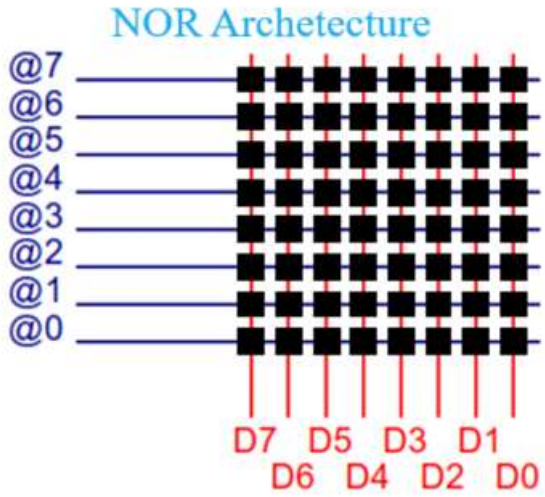


Figure 48: NOR Flash Architecture

Intel, the well-known processor manufacturer, has bet on this technology, because NOR flash memory has:

- the ability to interact directly with the processor;
- As a result, NOR memory is mainly used for storing programs that run directly ("XIP" or eXecute In Place):
    - Rarely modified
    - Computer BIOS
    - Firmware (OS), firmware of phones and cameras, etc.

Advantages

- Behavior of non-volatile RAM.
- Word-by-word programming and erasure possible.

- Low access time

Disadvantages

- Slow write/read per packet.
- Cost.

5.4.3.5.2- Flash NAND

NAND flash was developed by **Toshiba** in **1989.** The **NAND** architecture provides an assembly of the elementary cells of serial storage **with** the selection lines.

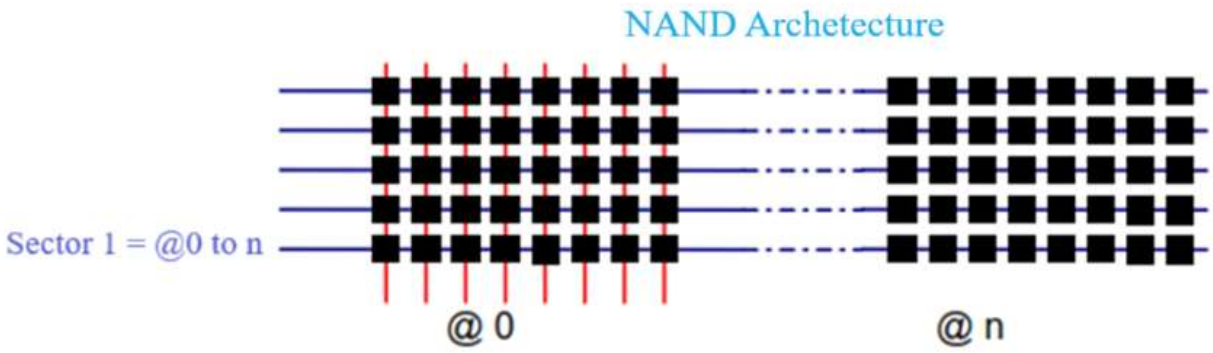


Figure 49: NAND Flash Architecture

Advantages:

- Behavior of non-volatile RAM.
- High integration density (low cost).
- Speed of write/read per packet
- Reduced consumption.

Disadvantages

- Unable to write/read per byte.
- Indirect I/O interface

5.4.3.5.3- Criteria for choosing between NOR Flash and NAND Flash

The major difference between NOR and NAND is their interfaces.

NOR :

- has dedicated address and data buses.
- 100% safe.

NAND :

- with an indirect I/O interface.
- is not 100% safe.

The main criteria to remember are:

- capacity

- speed
- consumption
- cost

An ideal memory would be one with a large capacity and a very short access time in order to be able to work quickly on this information. But it turns out that **high-capacity memories are often very slow,** and **fast memories are very expensive.**

In order to obtain the best cost-performance trade-off, a memory hierarchy is defined.
We use memories:

- Low capacity but very fast to store the information that the microprocessor uses the most.
- Large capacity but much slower to store the information that the microprocessor uses the least.

The further away you get from the microprocessor, the more the capacity and access time of the memories will increase.

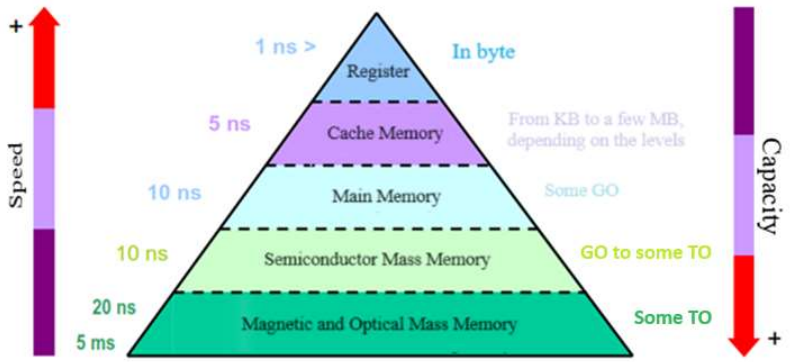


Figure 50: Memory hierarchy

- **Registers** are the fastest pieces of memory. They are located at the processor level. (SRAM)
- **Cache** memory is a fast, low-capacity memory designed to speed up access to central memory by storing the most used data. (SRAM)
- **The main memory** is the main organ for storing information. It contains the programs (instructions and data). (DRAM)
- **The supporting** memory serves as an intermediate memory between the core memory and the mass memory. It plays the same role as cache memory. (SRAM)
- **Mass** memory is a high-capacity peripheral memory used for the permanent storage of information (SSD, Hard disk, Flash Disk, DVD).

Chapter 6: Input/Output Interfaces

6.1- Introduction

The function of a microprocessor system is information processing. It is therefore obvious that it must acquire the information provided by its environment and reproduce the results of its treatments.

A complete system has two components, one hardware, the other software.

The Hardware component includes:

- The processor,
- Peripherals
- Buses allowing everyone to communicate.

The Software Component:

It can be summed up in the operating system, which provides the user with an abstract and simplified view of the operation of the hardware system and manages all of its resources.

Each system is therefore equipped with one or more I/O interfaces to ensure communication between the microprocessor and the outside world.

I/O techniques are very important for system performance. There's no point in having a microprocessor calculating very quickly if it often has to waste its time reading data or writing its results.

During an I/O operation, information is exchanged between the main memory and a device attached to the system. This exchange requires an interface (or controller) to manage the connection. Several techniques are used to carry out these exchanges.

6.2- The I/O Interface

Each device will be connected to the system through an interface (or controller) whose role is to:

- Connect the device to the data bus
- Manage exchanges between the microprocessor and the device

It is made up of:

- A control register in which the processor describes the work to be performed (transfer direction, transfer mode)
- One or more data registers that contain the words to be exchanged between the device and the memory
- A status register that shows if the exchange unit is ready, if the exchange went well, etc.

The interface data is accessed through an **I/O** address space.

6.3- Data Exchange Techniques

Before sending or receiving information, the microprocessor needs to know the status of the device. If a device is ready to receive or transmit information!!

To ensure that the transmission is done correctly. There are 2 ways to exchange information:

- The mode programmed by polling or interruption where the microprocessor serves as an intermediary between the memory and the peripheral
- Direct memory access (DMA) mode where the microprocessor is not responsible for data exchange.

6.3.1- Polling

The microprocessor queries the interface to see if transfers are ready. Otherwise, he waits. The major disadvantage is that the microprocessor often finds itself in the waiting phase. It is completely occupied by the input/output interface. The initiative of data exchange depends on the program executed by the microprocessor. This type of exchange is very slow.

6.3.2- Interruption

An interrupt is a signal, usually asynchronous to the current program, that can be emitted by any device external to the microprocessor. The microprocessor has one or more inputs dedicated to this purpose. Subject to certain conditions. It can interrupt the current work of the microprocessor to force the execution of a program that addresses the cause of the interruption.

An interruption can be initiated by:

1. **Hardware interrupt**: One of the electronic components of the CPU (e.g., keyboard, mouse, interface, printer, hard drive, etc.).
2. **Software Interrupt:** The program that is running.
3. **Exception:** An error in the running program

Note: An interrupt can be initiated by the processor itself in case of problems (division by zero, bad memory, etc.).

In interrupt data exchange, the microprocessor executes its main program until it receives a signal on its interrupt request line. It then takes care of the data transfer between the interface and the memory.

6.3.2.1- Operating principle of an interrupt

Before each instruction execution, the microprocessor checks to see if there has been a query on its interrupt line. If this is the case, it interrupts all these activities and saves the present state (registers, PCs, accumulators, status register) in a particular register called the battery (LIFO). Then, it runs the interrupt program and then returns to the saved state before resuming the main program.

Remarks

1. Some interrupt source has its own permission to operate in the form of a bit to be positioned, this is called the interrupt mask.
2. We can therefore prohibit or allow certain sources of interrupts, they are called **maskable interrupts**.
3. Each interrupt source has an interrupt vector where the starting address of the program to be executed is stored.
4. Interruptions are prioritized. In the event that multiple interrupts occur at the same time, the microprocessor first processes the one with the highest priority.

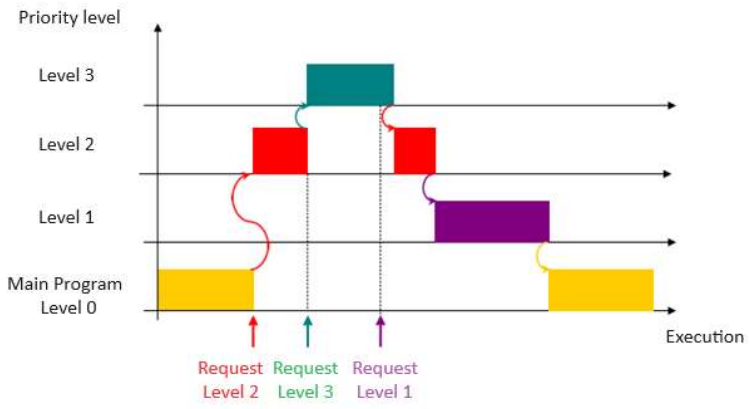


Figure 51: Principle of Interrupt Priority

The following diagram, Fig., summarizes the organization of interrupt vectors in the 8086 microprocessor.

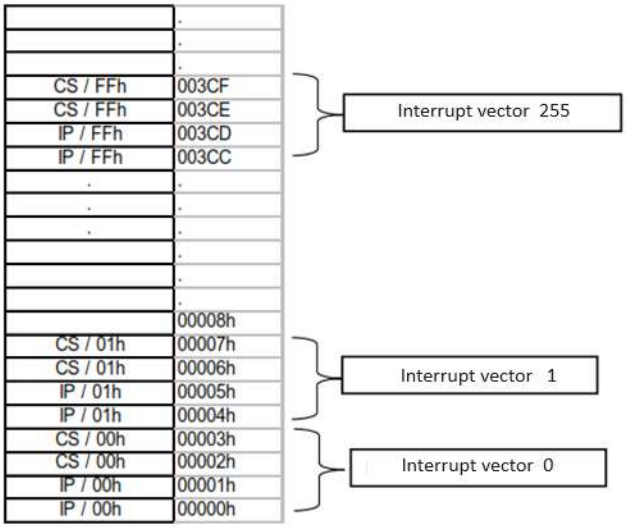


Figure 52: Organizing the 8086 Interrupt Vector Table

- **N° - Adresse - Fonction**
- **10 -** 040-043 - BIOS: Fonction vidéo
- **11 -** 044-047 - BIOS: Déterminer configuration
- **12 -** 048-04B - BIOS: Déterminer la taille mémoire de la RAM
- **13 -** 04C-04F - BIOS: Fonctions disquettes/disque dur
- **14 -** 050-053 - BIOS: Accès à l'interface sérielle
- **15 -** 054-057 - BIOS: Fonctions cassettes ou étendues
- **16 -** 058-05B - BIOS: Test du clavier
- **17 -** 05C-05F - BIOS: Accès à l'imprimante parallèle
- **18 -** 060-063 - Appel du BASIC en ROM
- **19 -** 064-067 - BIOS: Lancer système (ALT CTRL DEL)
- **1A -** 068-06B - BIOS: Lire date et heure

Figure 53: Exemples de type d'Interruption du 8086

6.3.4- Direct Exchange to Memory (DMA)

This mode allows the transfer of blocks of data between memory and a device without going through the microprocessor. To do this, a circuit called a **DMA (**Direct Memory Access) controller takes care of the various operations. The DMA takes care of the entire transfer of a block of data.

The microprocessor still must:

- Initiate the exchange by giving the DMA the identification of the affected device.
- Give the direction of transference.
- Provide the address of the first and last word involved in the transfer.

A DMA controller is equipped with:

- an address register,
- a data register,

- a meter
- and a control device (wired logic).

For each word exchanged, the DMA asks the microprocessor:

- Bus control,
- Reads or writes memory to the address in its register and releases the bus.
- It then increments this address and decrements its counter.
- Informs the processor of the completion of the transfer by a suspend line, when the counter reaches zero.

Advantage of DMA:

1. The processor is free to perform any kind of processing, for the duration of the transfer.

DMA Constraint:

Limiting its own memory access for the duration of the operation, since it sometimes has to delay some of its accesses.

Note : To allow the direct memory access device to perform its own: Cycle theft occurs.

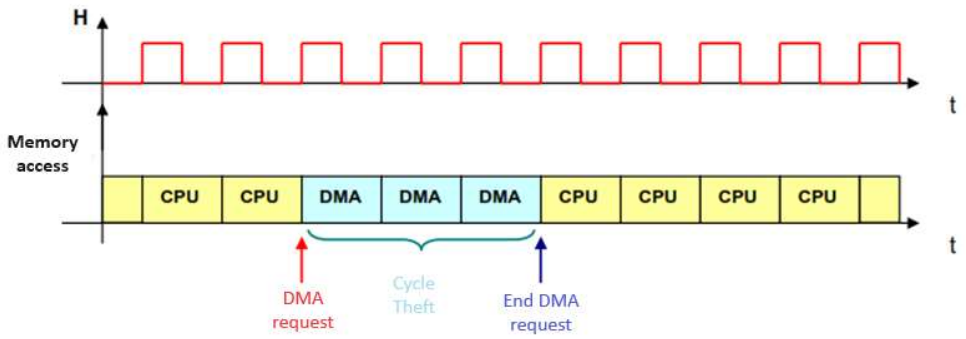


Figure 54: DMA Cycle theft

6.4- Types of Links

Microprocessor-based systems use two different types of links to connect to devices:

- Parallel link.
- Serial link.

A type of link is characterized by its **transmission rate** or **throughput** (in bit/s).

6.4.1- Parallel Link

All the bits of a word are transmitted simultaneously. The transmission is timed by a clock.

Advantage:

- Fast Transfers

Constraint:

- Limited to short transmission distances, due to:
  - Large number of lines required
  - cost
  - traffic congestion
  - Electromagnetic interference problems between each line (reliability).

Example of a parallel PC bus: The IDE, PCI, AGP bus (the latter two are replaced by the PCI Express).

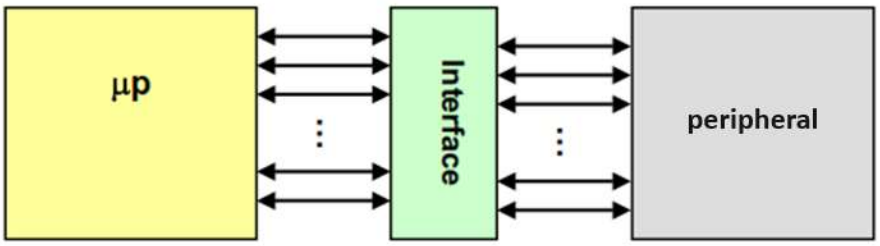


Figure 55:Principle of parallel link

6.4.2- Serial Link

In this type of connection, the bits constituting a word are transmitted one after the other on a single wire.

Advantage:

- Much longer transmission distances

Constraint:

- The transmission speed is slower

Example of a PC serial bus: SATA, USB, ...

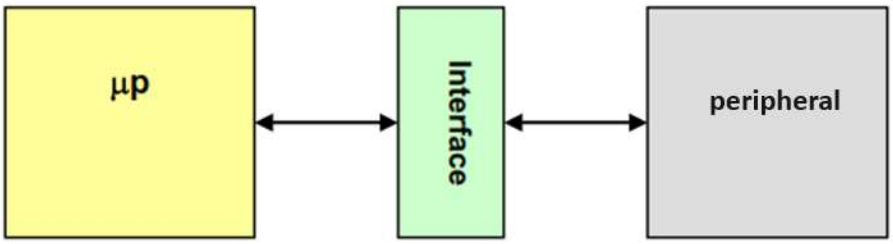


Figure 56: Principle of serial link

Serial data transmission can be conceived in two different ways:

- **Synchronous mode:** the transmitter and receiver have a synchronized clock which times the transmission.
    - The flow of data can be uninterrupted.

Asynchronous mode: transmission takes place at the rate of the presence of data.

- The characters sent are surrounded by a ***start*** signal and a ***stop signal***.

6.4.2.1- Asynchronous serial link

So that the communicating elements can understand each other, it is necessary to establish a transmission protocol. This protocol must be the same for each element. The parameters that come into play in this type of connection are:

The length of the transmitted words: 7 bits (ASCII code) or 8 bits

- Transmission rate: range from 110 bit/s to 128000 bit/s (determines the clock speed of the transmitter and receiver).
- Parity: the transmitted word may or may not be followed by a parity bit.
    - is used to detect possible transmission errors.
    - There are two types of parity:
        - **An even parity**, the total number of bits to 1 transmitted (including the parity bit) must be even.
        - **an odd parity, which is the inverse for an odd parity.**

The asynchronous serial link is initiated by a Start bit and terminates a Stop bit:

- **Start bit:** The line at rest is in state 1 (used to test a line cut). Moving to the bottom state of the line will indicate that a transfer is about to begin. This synchronizes the receive clock.
- **Stop bit:** After transmission, the line is set to level 1 for a certain number of bits in order to specify the end of the transfer. In principle, one, one and a half or 2 stop bits are transmitted.

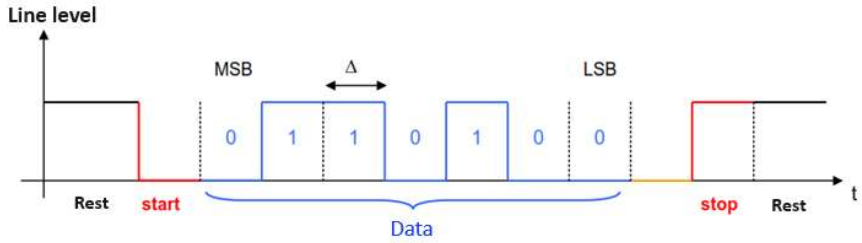


Figure 57: DMA Cycle Thefts

6.5- Architecture of a PC

The architecture of a PC can be summed up in the architecture of its motherboard, Fig.58:

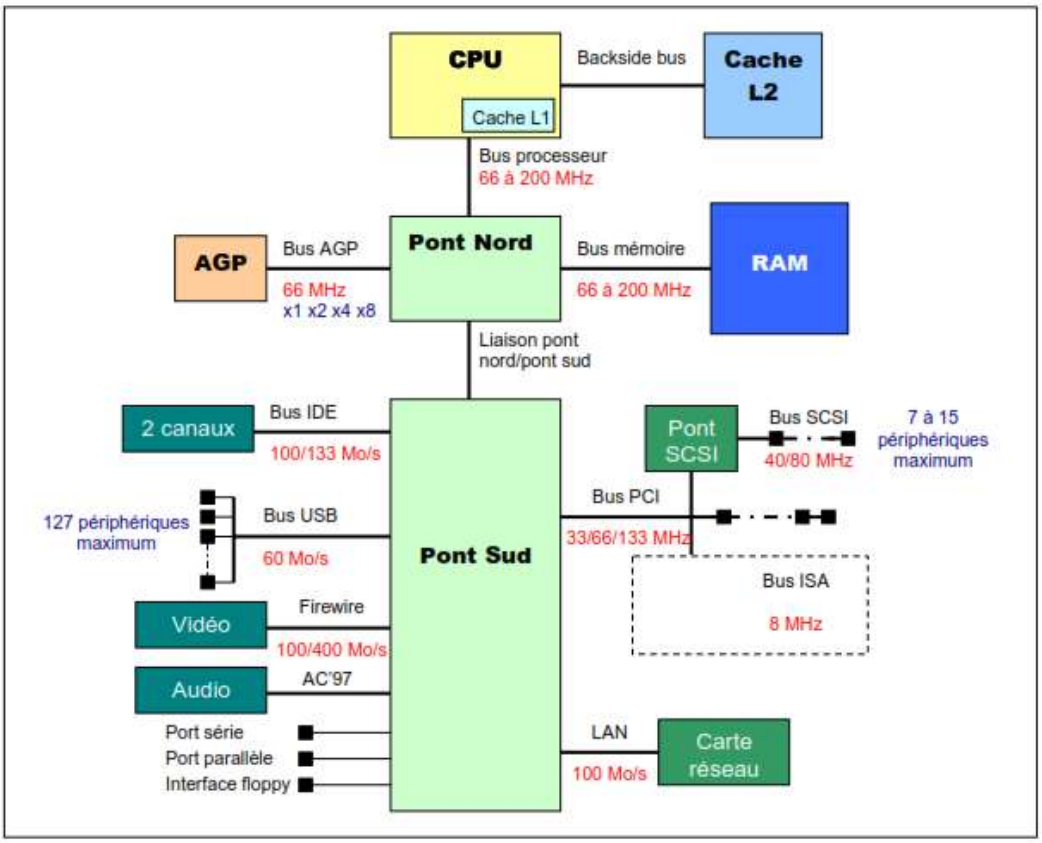


Figure 58: PC motherboard architecture

6.5.1- The chipset

It consists of a set of several components responsible for managing the communication between the microprocessor and the peripherals. This is the link between the different buses on the motherboard.

6.5.2- BIOS (Basic Input Ouput Service)

The BIOS is the program responsible for managing hardware: keyboard, monitor, hard drives, serial and parallel links, etc. It is stored in a read-only memory (EEPROM) and acts as an interface between the operating system and the hardware.

6.5.3- The Clock

It is used to synchronize the processing of instructions by the microprocessor or the transmission of information on the various buses.

6.5.4- Connection ports

They allow you to connect peripherals to the different buses of the motherboard. There are "internal" ports for connecting expansion cards (PCI, ISA, AGP) or storage devices (SCSI, IDE, Serial ATA) and "external" ports for connecting other devices (serial, parallel, USB, firewire, etc.)

6.5.5- The socket

This is the name of the connector for the microprocessor. It determines the type of microprocessor that can be connected.

Different buses are seen to transport information between the microprocessor and the memory or peripherals:

- Processor bus: it is also called a system bus or FSB (Front Side Bus). It connects the microprocessor to the north bridge and then to the memory. It's a 64-bit bus.
- IDE Bus (Integrate Drive Electronique)
- PCI Bus (Peripheral Component Interconnect)
- AGP Bus (Accelered Graphic Port)
- ISA Bus (Industry Standard Architecture)
- SCSI Bus (Small Computer System Interface)
- USB Bus (Universal Serial Bus)
- Firewire Bus: it's a serial SCSI bus
- Serial Ata (SATA) Bus, IDE bus replacement
- PCI Express Bus, replacing PCI and AGP buses
- Bluetooth
- WIFI (WIreless FIdelity Network)
- …

Lab Handouts

Lab 1: Getting Started with Assembler intel 8086

Objectives:

- Familiarize yourself with **the Emu8086** software.
- Get started with the **Assembler language**.

Material used:

- A PC.

I/ First steps in programming:

One of the basic operations that is performed in **8086** assembler programming is data transfer. This is done through the instructions: **MOV**.

I – 1 / The MOV instruction:

The **MOV statement** (from the word **Move**). In assembler, this statement transfers from a source location to a destination one:

MOV destination, source

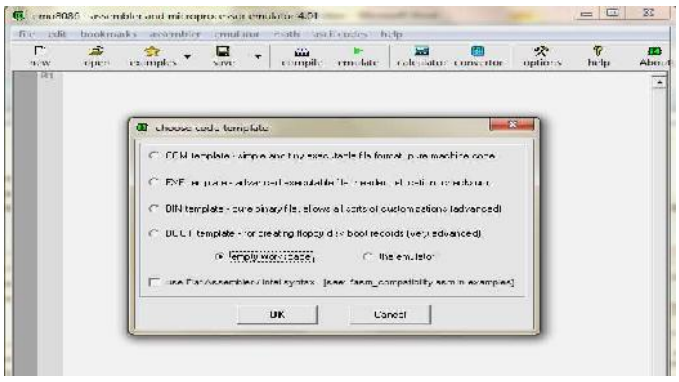
Possible transfers:

Destination	Spring
Register	Register
Register	Memory
Memory	Register
Register	Immediate value
Memory	Immediate value

I – 2 / Getting started with Emu8086 :

1 – Open the **emu8086 emulator:**

2 – Choose a new document by clicking on **New** in the displayed menu.

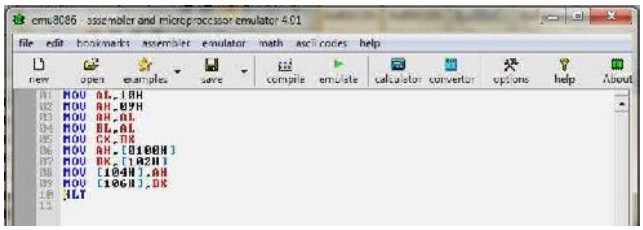


3- Next, click on empty **workspace** in the list displayed in order to have an empty document.

4- Enter the following assembly code:

```
MOV AL, 10h
MOV AH, 09h
```

5- Emulate the code by clicking on **emulate** on the main taskbar.



6- Two windows will be displayed. A window (**Original souce code**) contains the code you just wrote.

A second window (**emulator**) that will allow us to **execute** the code, see the **contents of the different** registers and many other things that we will see as we go along.

7 - Execute the code written in this way by clicking on **run**.

8 – What do you notice about the content of the different registers?

.....

.....

.....

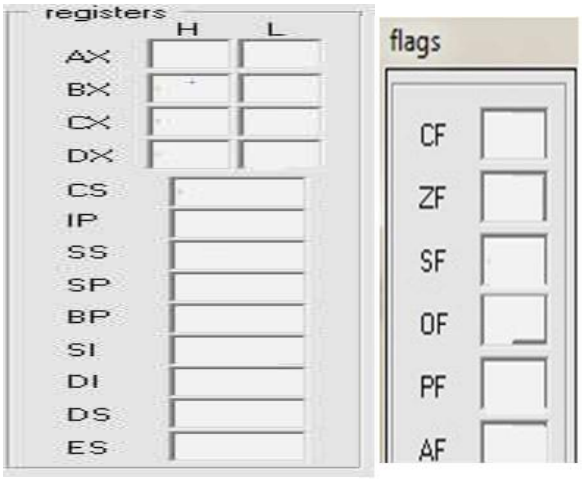
.....

.....

.....

9 - Now, on the emulator window, click on **reload** in order to reload the code (**i.e.** put it back in memory to re-execute it). Then click on **single step** several times and notice the changes in the registers each time.

10 - Give the results in the following table:



10 – What is the role of **Single Step?**

.....

.....

.....

11 – Register this code under the name: **myprog1.asm.**

I – 3 / Register and memory manipulation:

1 – Enter the following assembly code:

```
MOV AX, 05h
MOV [100h], AX
```

2 - Email the code by clicking on **emulate** on the main taskbar.

3 - Execute the code written in this way by clicking on **run**.

4– What do you notice about the content of the different registers?

.....

.....

.....

.....

.....

5– What do you notice about memory?

.....

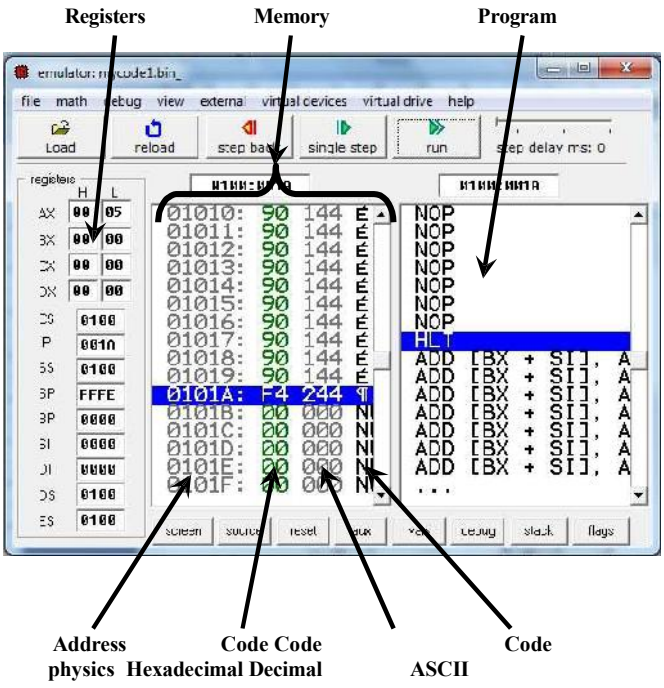
.....

.....

.....

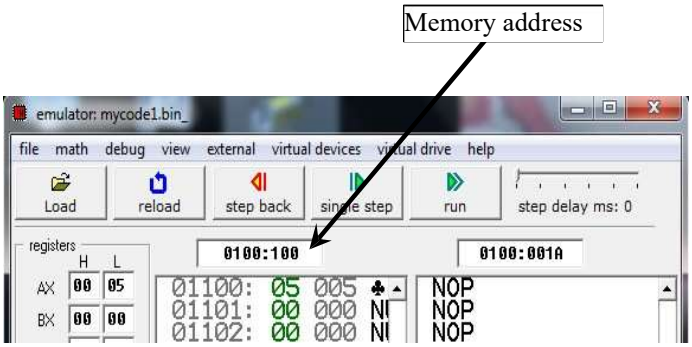
Indication:

After execution, we will have the following window (emulator window), several pieces of information are present: about the registers, the memory, about the program...



Notice that there are multiple lines of code that end in HLT, and the corresponding address is **0101A.** This is simply due to the fact that the program itself is stored in memory.

6– Now type 100 in the box indicating the memory address and press keyboard enter



7 – Now look at the box corresponding to 0100 and complete the table **.**

01100: 05 005 ♣

Value	Meaning
01100	
05	
005	
♣	

8- Enter the following assembly code:

```
MOV AX, 55h
MOV [100h], AX
```

9 - Emulate the code by clicking on **emulate** on the main taskbar.

10 - Execute the code written in this way by clicking on **run**.

11 – Enter the following assembly code:

```
MOV AH, 05h
mov [100],ah
```

12 - Emulate the code by clicking on **emulate** on the main taskbar.

13 - Execute the code written in this way by clicking on **run**.

14 – Complete the table below.

Value	Case 1	Case 2
Physical address		
Hexadecimal value		
Value in decimal		
ASCII Value		

15- What is the difference between using AX and AH?

.....

.....

.....

.....

16 – Comment:

.....

.....

.....

.....

.....

.....

16- Enter the next assembly code and do the same work again.

```
MOV AL, 55h
mov ah, 36h
MOV [100h], AL
MOV [104h], AX
```

17- Complete the following table:

Memory box	**101h/100h**	**105h/104h**
Hexadecimal value		
Value in decimal		
ASCII Value		

18 – Comment:

.....

.....

.....

.....

19- Enter the next assembly code and do the same work again.

```
MOV AX, 3655h
MOV [100h], AX
MOV BX, 100h
MOV CX, [BX]
```

20- Complete the following table:

Amended Records	Modified memory boxes
.....	.....
.....	.....
.....	.....
.....	.....

21 – Describe what each instruction does.

instruction	**Description**

22- Enter the next assembly code and do the same job again.

```
MOV [100h], 36h
MOV [104h], 52h
MOV [106h], 69h
MOV BX, 100h
MOV CX, [BX]
MOV DX, [BX+4]
MOV AX, [BX]+6
```

23- Completing the following table:

Amended Records	Modified memory boxes
.....	.....
.....	.....
.....	.....
.....	.....
.....	.....
.....	.....
.....	.....
.....	.....

24 – Comment:

.....

.....

.....

.....

.....

.....

25 – Describe what each instruction does.

instruction	Description

26 – Conclusion and general remarks on Lab1:

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

Lab 2: Arithmetic Instructions in 8086

Objectives :

- Become familiar with the set of arithmetic instructions.

Material used:

- A PC.
- Emul8086 **software.**

Note: -To consult the instruction set, click on the "help" tab of the emulator, a browser window opens:
- Choose "8086 Instruction Set"
- Then choose the instructions you want to use.

I/ Theoretical preparation:

A/ Reminder:

The **8086 is a 16-bit** microprocessor**, so** it's not appropriate **to have 20-bit memory addresses** [Or 5 digits in hexadecimal representation] (**This is mentioned because the technology has produced** 20-bit address buses). The solution adopted is:

- Divide the memory into pages (called **segments**).
- Represent a 16-bit (or 4-digit hexadecimal memory) **relative address (offset)** traversing a segment.

So instead of representing an address by **20350, we use** 2000 instead: 350 or: **2000 is the** segment and **350** is the offset.

Analogy to the numbering used in hotels. When booking room **213 ..**

It is immediately clear that this is bedroom **13** on floor **2.**

(Then stage **2** represents **the segment** and stage **13** represents **the offset**. Then we can write **2:13**)

B / Give the absolute (physical) addresses of the following addresses:

Logical Address	Absolute Address
3500 : A600	
1036: FFF0	
2000 : 0350	
3C47: 3190	

Your remarks/comments on the results of the table:

.....

.....

.....

C / Indicate the correct instructions and correct the others:

Instruction	Correction
MOV AX, 9h	
MOV 9h, AL	
MOV AH, BX	
MOV AX, [BX]	
MOV AL, [BX+2]	
MOV AX, N1	
MOV AX, [1]	
MOV AX, BL	

II / Arithmetic operations in 8086:

II.1/ Unsigned numbers:

1 – Enter the following assembly code:

```
MOV AX, 05h
MOV BX, 3 p.m.
ADD AX, BX
HLT
```

2-Run the program in step-by-step mode and complete the following table:

Instruction	Regis. modified by the instruction	Result (value)

4- Give your comments on the program:

.....

.....

.....

5 – Now enter the following assembly code:

```
MOV AX, 195h
MOV BX, 911h
ADD AX, BX
HLT
```

6- Complete the following table:

Instruction	Regis. modified by the direction	Result (value)

7 – Now enter the following assembly code:

```
MOV AX, 195h
ADD AL, 02h
HLT
```

8- Complete the following table:

Instruction	Regis. modified by the direction	Result (value)

9 – Your remarks between example **5** and **7**:

.....

.....

.....

10– Now enter the following assembly code:

```
MOV AX, 1002h
INC AH
MOV BX, 0200h
DEC BH
SUB AX, BX
SUB AH, 03H
HLT
```

11- Complete the following table:

Instruction	Regis. modified by the direction	Result (value)

12 – Now enter the following assembly code:

```
DW number?
MOV AX, 04h
MOV number, 05h
MUL number
HLT
```

13– What does "number" represent?

.....

.....

14 – Complete the following table:

Instruction	Regis./Memory Box Modified by the Instruction	Result (value)

15 – Give the final result in decimal and comment on:

.....

.....

.....

16- Now enter the following assembly code:

```
DW number?
MOV AX, 5F4h
MOV Number, 99h
MUL number
HLT
```

17 – Complete the following table:

Instruction	Regis./Memory Box Modified by the Instruction	Result (value)

18 – Give the final result in decimal and comment on:

.....

.....

.....

19- Now enter the following assembly code:

```
MOV AX, 8 p.m.
MOV BL, 05h
DIV BL
HLT
```

20 – Complete the following table:

Instruction	Regis./Memory Box Modified by the Instruction	Result (value)

21 – Give the final result in decimal places and comment on:

.....

.....

.....

22- Now enter the following assembly code:

```
DW number?
MOV AX, 0F04h
MOV DX, 35ECh
MOV number, 9BCDh
DIV number
HLT
```

23 – Complete the following table:

Instruction	Regis./Memory Box Modified by the Instruction	Result (value)

24 – Give the final result in decimal and comment on:

.....
.....
.....

24- Conclusion and general remarks on Lab2:

.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....

Lab3: Branching instructions and comparison in 8086 (loop concept)

Objectives :

- ✓ Familiarize yourself with the set of conditional and unconditional branching instructions and comparison (loop concept).

Material used:

- A PC.
- Emul8086 software.

1- **Reminder:** Jumps play a very important role in programming. There are two types of jump instructions. Unconditional jump statements, a single "JMP" statement and a set of conditional jump statements, JE, JNE, JB, JBE, JA, JAE, ..., with which we add the "LOOP" statement see help in the software.

2- Let the following assembler code be used:

```
org 100h
T dw 10h,20h,30h,40h,50h,60h,70h,80h,90h,100h
N dw 10h
S dw?

        MOV BX, 0h
        MOV AX, 0h
ETQ:    ADD AX, T+BX
        INC BX
        INC BX
        CMP BX, N
    JL etq
        MOV S, AX
        HLT
```

3- **Run** the program, and then complete the following table:
(for a good understanding of the program, run it in step-by-step mode and follow the evolution of the result in a table)

Reg/Var	Value after each step									
	0	1	2	3	4	5	6	7	8	9

4- What does this program do?

.....

.....

.....

5- Let the following assembler code be used:

```
org 100h
T dw 10h,20h,30h,40h,50h,60h,70h,80h,90h,100h
N dw 10h
S dw?
        MOV DX, 0h
        MOV CX, N
        MOV AX, 0h
ETQ:    ADD AX, T+BX
        INC BX
        INC BX
        LOOP etq
        MOV S, AX
        HLT
```

6- **Run** the code: for a good understanding of the program, run it in step-by-step mode and follow the evolution of the result in a table)

Reg/Var	Value after each step									
	0	1	2	3	4	5	6	7	8	9

7- What does this program do?

.....

.....

.....

8- Your remarks and conclusion between example 2 and 5:

.....

.....

.....

9- **Conclusion and general remarks on Lab3:**

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

Lab 4: Interruptions and the Stack in 8086

Objectives :

- Stack Usage
- Use of Software Interrupts.

Material used:

- A PC.
- Emul8086 **software.**

I/ Theoretical preparation :
Has/

Reminder 1: The Stack

The stack is an area of memory that allows you to quickly store and retrieve values. A stack works like a stack of real objects, in **LIFO** (Last In First Out) mode. The maximum number of values is set at the beginning. A piece of data in the stack is determined by two parameters:

- The SS Stack Segment ;
- The Stack Pointer (SP Offset).

There are two things you can do on the stack:

- Add **a value (16 bits)** to the top of the stack. In this case, we use the statement:

  PUSH Register (16-bit)

  The contents of the registry in question are loaded into the SS: SP box**, and then:** SP becomes SP-2**.**

- Remove **the value (16 bits)** at the top of the stack (unstack). In this case, we use the statement:

  POP Register (16-bit)

  The contents of the SS:SP **are loaded into the registry in question, and then:** SP becomes SP+2**.**

Reminder 2: Interruptions

An interrupt, as the name suggests, interrupts **the normal execution of program instructions to** execute others before returning to normal operation.

An interruption can occur in two ways:

9- **It can be used by the hardware** to perform processing (reading a key on the keyboard, for example). In this case, the device triggers an IRQ (Interrupt ReQuest) that has a number specific to the port to which the device is connected. Each IRQ corresponds to a particular barrier. This is a **hardware interrupt** (**it will not be covered here**).

10- **It can be requested by the software** running with the INT statement. This is a software interruption, which will be our object.

Software Interruptions:

11- Each trap has a number between 0 and 255 because it is encoded in 1 byte.

12- The INT statement has a single operand: the number of the interrupt to be called, but it can have multiple subinterrupts. The sub-interrupt number is denoted by a value (h) loaded into the register **AH :**

INT *val (hexadecimal).*

- Each interrupt corresponds to a routine whose memory address is stored in the **interrupt vector table**. This table is stored in memory at address **0000:0000**, and stores for each vector, the address of the routine to be called, in 4 bytes (offset then segment, each in 2 bytes).

II/ Operation in 8086

Part A:

1- Enter the assembly code:

```
MOV     AH, 01h
INT     9 p.m.
HLT
```

2- Emulate the program.
3- Open the "emulator screen" window, by clicking on the "screen" button in the "Emulator" window.
4- Arrange the windows so that they are all visible.
5- Run the program in single step mode
6- What do you notice on the "emulator screen" when you execute the 'INT 21h' statement?

.....

.....

7- Press a key on the keyboard.
8- Repeat the program 5 times. Each time you change the character you type. Complete the following table:

N°	Selected keyboard key	ASCII Hex Character Code	Value in Reg. AL
1			
2			
3			
4			
5			

9- Comment 8, after consulting the ASCII code table.

.....

.....

10- Your comments on the program (assembly code 1):

.....

.....

.....

11- Enter the assembly code:

```
MOV     AH, 02h
MOV     DL, 42h; 46h 4Bh 3Dh 7Ah
INT     9 p.m.
HLT
```

12- Emulate the program
13- Run the program 5 times, each time you change the value in 'DL' to one of the values that are after the ';'
14- What do you notice on "emulator screen", every time you run the program

.....

.....

15- Complete the following table:

N°	Value in Reg. 'DL'	Displayed in 'Emulator Screen'	ASCII Hex Character Code	Value in Reg. 'AL'
1	42 hours			
2	46 hours			
3	4Bh			
4	3Dh			
5	7Ah			

16- Comment 15, after consulting the ASCII code table.

.....

.....

17- Your comments on the program (assembly code 11):

.....

.....

.....

Part B:

1- Enter the following assembly code:

```
MOV     AH, 2Ah
INT     9 p.m.
HLT
```

2- Run the "run" program.

3- Complete the following table:

Register	Contents (hex)	Contents (dec)
LY		
DL		
DH		
CX		

4- In interpreting the results in Table 3, what are your observations? And what does this program do?

.....

.....

.....

.....

.....

5- Enter the following assembly code:

```
MOV     AH, 2ch
INT     9 p.m.
HLT
```

6- Run the "run" program.

7- Complete the following table:

Register	Contents (hex)	Contents (dec)
CH		
CL		
DH		
DL		

8- In interpreting the results in Table 7, what are your observations? And what does this program do?

.....

.....

.....

.....

.....

Part C:

1- Let the following assembler program be used:

```
        MOV AH,01h
        MOV CX,08h
Q1:     INT 9pm
        PUSH AX
        LOOP Etq1
        HLT
```

2- Emulate the program.

3- Open the "emulator screen" window, by clicking on the "screen" button in the "Emulator" window.

4- Open the "Stack" window, by clicking on the "stack" button in the "Emulator" window.

5- Arrange the windows so that they are all visible.

6- Run the "run" program

7- Type the word "TELECOM!" in "emulator screen".

8- Complete the following table:

SS	SP	Content		ASCII code of L content
		H	L	

9- What is the role of SP?

.....

.....

10- How does it work (SP)?

.....

.....

11- Looking at Table 8, comment on Program 1:

.....

.....

.....

.....

12- **Note:** The following program code allowed you to move to the next line in "emulator screen":

```
MOV AH,02h
MOV DL,0Dh
INT 9pm
MOV DL,0Ah
INT 9pm
```

13- Fill in program code 1 so that the second line shows the inverted "TELECOM!" channel, but always keeping the "!" at the end. The result will be "MOCELET!", using the data from the stack.

14- Complete the following table after you have executed the program you have carried out.:

SS	SP	Content		ASCII code of L content
		H	L	

15- In analyzing Tables 8 and 14, provide your remarks:

.....

.....

.....

.....

.....

.....

.....

.....

Conclusion and general remarks on Lab 4:

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

Biblipgraphy

1- John P. Hayes, “Computer Architecture and Organization”, 2nd edition, McGraw -Hill Book Company, International Edition 1988.
2- Harold Lorin, “Introduction to Computer Architecture”, 2nd edition, Wiley-Interscience Publication”, 1989.
3- A. J. van de Goor, « Computer Achitecture and Design”, Addison-Wesley, 1989.
4- William D. Murray, “Computer and Digital System Architecture”, Prentice-Hall International Editions, 1990.
5- Datasheet Intel 8086.