People's Democratic Republic of Algeria
Ministry of Higher Education and Scientific Research
University of 20 août 1955 – Skikda

Faculty of Sciences
Department of Computer Science

Course handout:

Computer Architecture

Dr. Salah BOUGUEROUA

The course is intended for second-year undergraduate students of Computer Science

Academic Year: 2024/2025

Course educational plan

This manuscript is a course material for the subject "Computer Architecture." It is intended for second-year undergraduate students in Computer Science, in the Computer Science Department of the University of 20 août 1955 – Skikda.

The weekly hourly volume and the evaluation mode are given in the following table:

Course title		**Computer Architecture**
Domain		**Mathematics and Computer Science**
Branch		**Computer Science**
Weekly hourly volume	Lecture	**1h :30**
	Tutorial (TD)	**1h :30**
	Lab work (TP)	**1h :30**
	Personal work	**3h :00**
Coefficient		**3**
Credits		**5**
Assessment mode	Continuous	**40 %**
	Exam	**60 %**

Contents

COURSE EDUCATIONAL PLAN **2**

CONTENTS **3**

LIST OF FIGURES **5**

LIST OF TABLES **7**

GENERAL INTRODUCTION **8**

CHAPTER 1. INTRODUCTION TO COMPUTER ARCHITECTURE 9
1.1 Introduction to the Concept of Computer Architecture *10*
1.2 The von Neumann and Harvard Machines *11*
CHAPTER 2. MAIN COMPONENTS OF A COMPUTER 15
2.1 The Arithmetic and Logic Unit (ALU) *16*
2.2 Notions about buses *23*
2.3 Memory hierarchy *29*
2.4 Registers *30*
2.5 Memory *31*
2.6 Cache Memory *35*
CHAPTER 3. NOTIONS ON COMPUTER INSTRUCTIONS 44
3.1 High-level language, assembler, machine language *45*
3.2 Notions on instruction sets *47*
3.3 Addressing modes *48*
3.4 The different stages of generating an executable program *49*
3.5 Compilation *49*
3.6 Assembling process *50*
3.7 Control unit *51*
3.8 Steps for executing an instruction *52*
3.9 UCC Pipeline *53*
3.10 Clock *56*
3.11 Sequencer *56*
CHAPTER 4. PROCESSOR 59
4.1 Processor role *60*
4.2 Performance *60*
4.3 Measuring performance *60*
4.4 Cycle per instruction (CPI) *60*
4.5 CPU Execution Time *61*
4.6 Notions on CISC and RISC architectures *62*
4.7 Presentation of the MIPS R3000 Microprocessor *63*
4.8 External structure of the MIPS R3000 processor *63*
4.9 Internal structure of the MIPS R3000 processor *68*
4.10 MIPS R3000 instruction set *69*
4.11 MIPS Processor Addressing Modes *75*
4.12 Programming in MIPS Assembly Language *77*
CHAPTER 5. SPECIAL INSTRUCTIONS 81
5.1 Concepts about interruptions and exceptions *82*
5.2 Input/Output *84*
5.3 Special Instructions *92*
5.4 System Control Coprocessor (CP0) Instructions *92*

APPENDIX: PRESENTATION OF THE MARS SIMULATOR **94**

BIBLIOGRAPHY 96

List of figures

Figure 1.1: A six-level computer [1] 11
Figure 1.2 : Harvard Architecture 12
Figure 1.3 : IAS machine 12
Figure 1.4 : EDSAC 13
Figure 1.5 : Original von Neumann machine 13
Figure 2.1 Symbolic Representation of ALU 16
Figure 2.2 : Some examples of the 2's complement addition operation. (a-d): Correct sums, (e, f): Incorrect sums (overflow). The final carry (the 1 in gray) can be ignored (b, d, e). 18
Figure 2.3 : Some examples of the 2's complement subtraction operation (Note the conversion from subtraction to addition). (a-d): correct results, (e, f): incorrect results (overflow). The final carry (the 1 in gray) can be ignored (a, c, e). 19
Figure 2.4: The different fields and their sizes of the IEEE 754 standard. (a): Single precision, (b): Double precision 20
Figure 2.5 : Representation of the binary number (101.11) according to IEEE 754 standard 20
Figure 2.6: Half adder and its symbolic representation 22
Figure 2.7: Full adder and its symbolic representation 22
Figure 2.8: Example of eight-bit ripple-carry adder 22
Figure 2.9: A single-bit Arithmetic Logic Unit [1] 23
Figure 2.10: Illustrative image of a bus 24
Figure 2.11: Bus types 24
Figure 2.12: Conceptual division of wires that comprise a bus into lines for control, addresses, and data 25
Figure 2.13: Parallel and serial transfer 25
Figure 2.14: Full-duplex and half-duplex transmission 26
Figure 2.15: Logical structure of a simple personal computer [1] 27
Figure 2.16: Illustration of a bridge connecting two buses 28
Figure 2.17: Architecture of an early Pentium system. [1] 28
Figure 2.18: Data multiplexing 29
Figure 2.19: Memory hierarchy 30
Figure 2.20: General diagram of a memory 32
Figure 2.21: Example of increasing word size using 2 ICs. 34
Figure 2.22: Example of increasing the number of words using 2 ICs. 34
Figure 2.23: Illustrative image of cache and main memory 35
Figure 2.24: Structure of cache and main memory 37
Figure 2.25: Illustrative image of direct-mapped cache memory 38
Figure 2.26: Main memory address format for direct mapping 38
Figure 2.27: Searching for a word in a direct cache 39
Figure 2.28: Illustrative image of the associative mapping 39
Figure 2.29: Main memory address format for associative mapping 40
Figure 2.30: Illustrative image of set-associative mapping 40
Figure 2.31: Main memory address format for set-associative mapping 41
Figure 2.32: Write through (image from Wikipedia) 42
Figure 2.33: Write back (image from Wikipedia) 43
Figure 3.1: Illustration of some addressing modes 48
Figure 3.2: Program production line 49
Figure 3.3: Assembling process 50
Figure 3.4: Central processing unit main components with the memory 51
Figure 3.5: Difference between sequential processing (a), and pipeline processing (b) 53
Figure 3.6: Second pipeline representation; the vertical axis represents the subtasks. 54
Figure 3.7: Illustrative image of a clock cycle 56
Figure 3.8: The sequencer 56
Figure 3.9: General structure of the microprogrammed sequencer [6] 57

FIGURE 3.10: MICROCOMMANDS 58
FIGURE 4.1: ILLUSTRATIVE DIAGRAM OF A MEMORY WORD 65
FIGURE 4.2 : DIFFERENCE BETWEEN LITTLE-ENDIAN AND BIG-ENDIAN 65
FIGURE 4.3: GENERAL MEMORY LAYOUT FOR A PROGRAM 66
FIGURE 4.4: SIMPLIFIED DIAGRAM OF THE MIPS DATA PATH [17] 67
FIGURE 4.5: MIPS R3000 PROCESSOR INTERFACE 68
FIGURE 4.6: THE INTERNAL ARCHITECTURE OF THE MIPS R3000 PROCESSOR [20] 69
FIGURE 4.7: ILLUSTRATION OF THE FIVE ADDRESSING MODES OF THE MIPS PROCESSOR 76
FIGURE 4.8: .DATA AND .TEXT SECTIONS OF AN ASSEMBLY PROGRAM 79
FIGURE 4.9: EXAMPLE OF DECLARATIONS OF SOME DATA OF DIFFERENT TYPES 80
FIGURE 5.1: READING AN INTEGER 85
FIGURE 5.2: READING A REAL NUMBER 85
FIGURE 5.3: READING A STRING 86
FIGURE 5.4: CODE FOR READING A CHARACTER STRING 86
FIGURE 5.5: PRINTING AN INTEGER 87
FIGURE 5.6: CODE FOR PRINTING A REAL NUMBER 87
FIGURE 5.7: PRINTING A REAL NUMBER 88
FIGURE 5.8: CODE FOR PRINTING A STRING OF CHARACTERS 88
FIGURE 5.9: PRINTING A STRING OF CHARACTERS 89

List of tables

TABLE 1.1: COMPARISON BETWEEN VON NEUMANN AND HARVARD ARCHITECTURES 14
TABLE 2.1: REPRESENTATIONS OF UNSIGNED NUMBERS (4-BIT EXAMPLE) 17
TABLE 2.2: DIFFERENT REPRESENTATIONS OF SIGNED INTEGERS (4-BIT EXAMPLE) 17
TABLE 2.3: REPRESENTATION OF PARTICULAR VALUES IN IEEE 754 20
TABLE 2.4: CHARACTERISTICS OF SOME BUSES 26
TABLE 3.1: PIPELINE DEPTHS OF SOME PROCESSORS 55
TABLE 4.1: COMPARISON BETWEEN VAX-11 (CISC) AND BERKELEY RISC-1 (RISC) 62
TABLE 4.2: GENERAL-PURPOSE REGISTERS OF THE MIPS R3000 PROCESSOR 63
TABLE 4.3: MIPS R3000 PROCESSOR REGISTERS USED FOR INTERRUPT AND EXCEPTION HANDLING 64
TABLE 4.4: MIPS INSTRUCTION ENCODING (OPCOD FIELD: BITS 31:26) 70
TABLE 4.5 : MIPS INSTRUCTION ENCODING (FUNCTION CODE FIELD: BITS 5:0) 70
TABLE 4.6: MIPS INSTRUCTION ENCODING (OPCOD FIELD = BCOND) 70
TABLE 4.7: MIPS INSTRUCTION ENCODING (OPCOD FIELD = COPRO) 70
TABLE 4.8: SOME ASSEMBLER DIRECTIVES 78
TABLE 5.1: SEVEN TYPES OF EXCEPTIONS RECOGNIZED BY THE MIPS R3000 PROCESSOR 82
TABLE 5.2: THE TOP TEN MIPS ASSEMBLY LANGUAGE SERVICES AND THEIR CODES 84

General introduction

Computer Architecture studies the structure and organization of computer systems and how their components interact to execute programs efficiently. This course introduces the main components of a computer system, including the processor, main memory, cache memory, and system buses. It also provides fundamental notions of computer instructions, the compilation and assembly process, and instruction execution mechanisms such as the instruction cycle and pipelining. Emphasis is placed on the MIPS processor, which is used as a case study to examine its instruction set, explore both its external and internal structure, and practice assembly-level programming.

The courses *Machine Structure 1* and *Machine Structure 2*, studied in the first year, constitute recommended prerequisite knowledge for this course.

The objective of this course is to clarify the functioning of a computer with a detailed presentation of computer architecture. To this end, this course is structured over five chapters. The first chapter is devoted to the presentation of some notions of computer architecture with a particular focus on von Neumann and Harvard machines. The second chapter is dedicated to the study of the main components of a computer such as the arithmetic and logic unit, buses, memories, etc. Among other things, we will present in the third chapter some notions on the instructions of a computer, the principle of compilation and assembly and the pipeline. Chapter four is generally devoted to the MIPS R3000 processor. Finally, notions on interrupts and I/O instructions and system instructions will be presented in the fifth chapter.

Chapter 1. Introduction to Computer Architecture

1.1 INTRODUCTION TO THE CONCEPT OF COMPUTER ARCHITECTURE 10
1.1.1 Definitions 10
1.1.1.1 Definition of "informatics»: 10
1.1.1.2 What is a computer? 10
1.1.1.3 Definition of "Computer Architecture»: 10
1.1.2 Layered Architecture 10
1.2 THE VON NEUMANN AND HARVARD MACHINES 11
1.2.1 Harvard Architecture 11
1.2.2 von Neumann Architecture 12
1.2.2.1 Key Components of von Neumann Architecture 13
1.2.3 Difference between von Neumann architecture and Harvard architecture 14

1.1 Introduction to the Concept of Computer Architecture

Throughout their career, a computer scientist will be required to write programs of varying complexity, optimize these programs to make them faster and more efficient, configure and maintain computer systems, develop compilers, among other tasks. These activities require a solid understanding of computer organization and operation. In particular, programming compilers demands an in-depth knowledge of the target hardware architecture to ensure that the generated code functions optimally on the intended systems.

1.1.1 Definitions

1.1.1.1 Definition of "informatics»:

The word "informatics" (French: informatique) is a contraction of "information" and "automatic."

Informatics (Computer science) is the science of automatic information processing.

"The science and set of techniques for collecting, sorting, storing, transmitting, and using information processed automatically with software programs implemented on computers."[1].

1.1.1.2 What is a computer?

"An electronic calculator equipped with high-capacity memory and high-speed information processing capabilities, capable of solving complex arithmetic and logical problems through the automatic execution of stored programs." [1].

A computer has the ability to acquire information, store it, process it, and then return it in another form:

1. It can receive input data → "Input function",
2. Perform operations on this data according to a program → "Processing function",
3. Finally, provide results as output → "Output function"

Programs that execute these operations can only run in binary language.

1.1.1.3 Definition of "Computer Architecture»:

Computer architecture studies how to design the parts of a computer system that are visible to programmers. (Tanenbaum, 2006)

Computer architecture focuses on structure and behavior, including elements such as instruction sets and formats, data types, the number and types of registers, etc.

1.1.2 Layered Architecture

A computer is built as a series of levels, each level designed upon its predecessors. Each level corresponds to a unique abstraction with specific objects and operations (Figure 1.1).

[1] Translated from *Le Grand Robert* dictionary

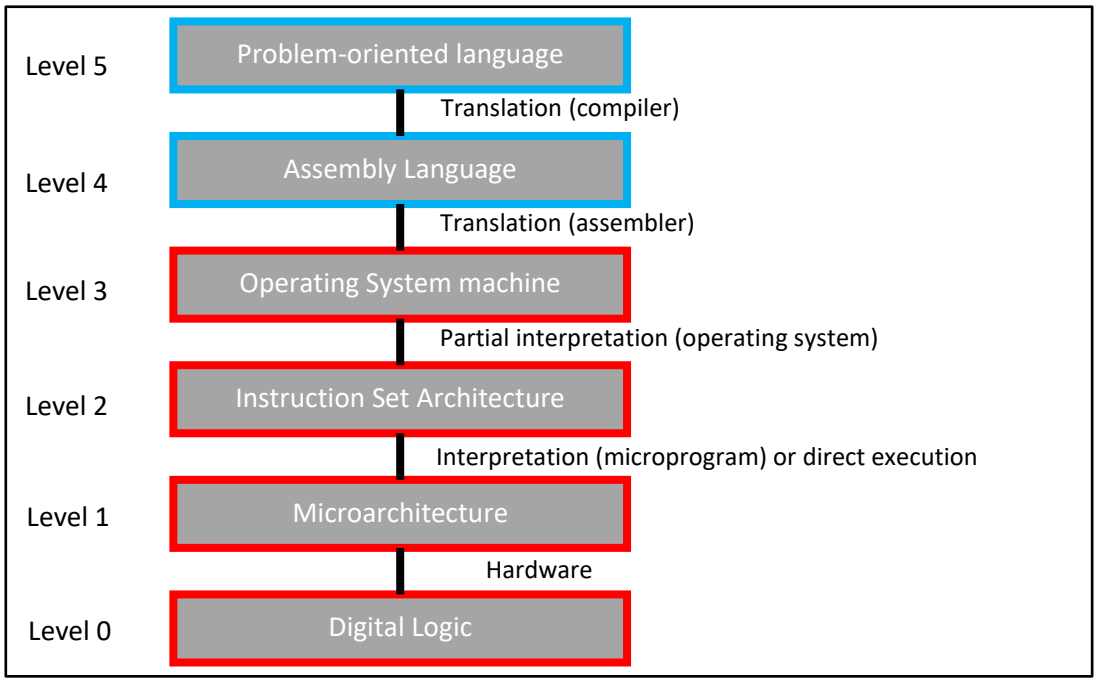


Figure 1.1: A six-level computer [1]

- **Digital Logic (Level 0)**: The actual hardware of the machine that executes programs in machine language (Level 1).
- **Microarchitecture (Level 1)**: Involves registers forming local memory and the arithmetic and logic unit, capable of performing simple arithmetic operations. Data flow is controlled by a "microprogram" or directly by hardware.
- **Instruction Set Architecture (Level 2)**: The interface between software and hardware, containing all the machine-executable instructions (e.g., Load, Store, Move, arithmetic instructions).
- **Operating System (Level 3)**: Provides system-level facilities such as virtual memory, file input/output, and parallelism.
- **Assembly Language (Level 4)**: A symbolic method for writing programs for levels 1, 2, and 3. Assembly programs are first translated to machine language and then interpreted by the appropriate physical or virtual machine.
- **Application Language (Level 5)**: High-level programming languages such as C, C++, Java. These programs are compiled into Level 3 or 4 code by a compiler.

1.2 The von Neumann and Harvard Machines

1.2.1 Harvard Architecture

In 1944, Howard Aiken of Harvard University created the ***Mark I*** [2]. This machine featured two separate memory units, one for instructions and another for data, leading to the "Harvard architecture" (Figure 1.2).

The advantage of this design is that instructions and data can be accessed simultaneously, improving performance.

[2] Also called *IBM Automatic Sequence Controlled Calculator* (ASCC)

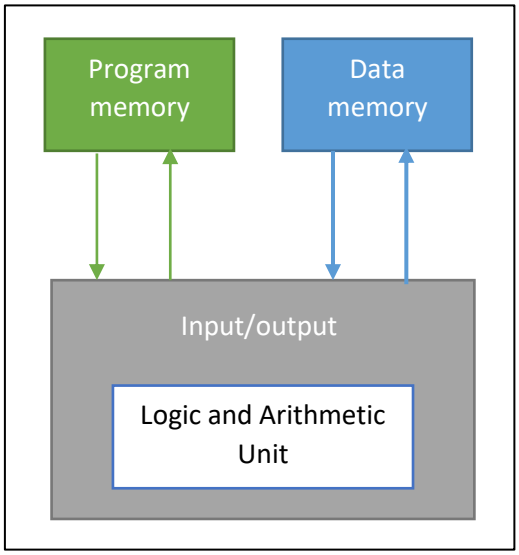


Figure 1.2 : Harvard Architecture

1.2.2 von Neumann Architecture

In 1945, von Neumann, a Hungarian mathematician, proposed a new architecture based on the "stored-program" concept. In 1946, he and his colleagues began designing a new computer based on this concept, called the ***IAS*** computer (Figure 1.3), at the Institute for Advanced Study in Princeton. The IAS computer was completed in 1952. It was the prototype for all subsequent general-purpose computers.

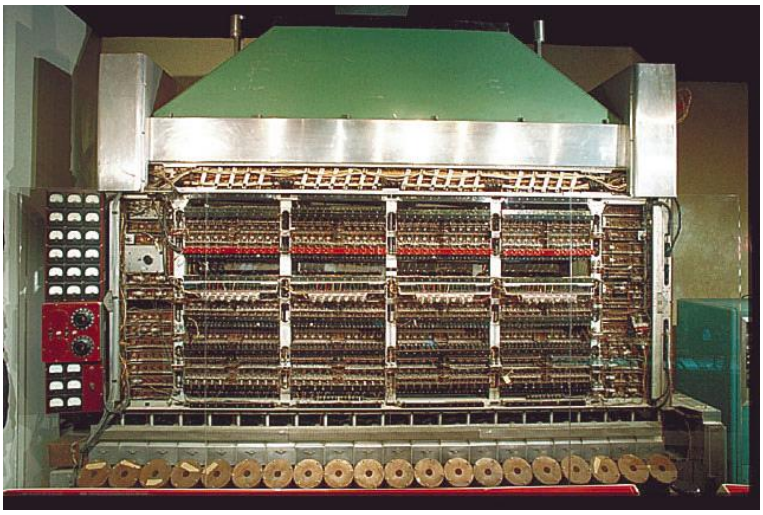


Figure 1.3 : IAS machine

The basic design that von Neumann first described was used in the EDSAC[3] (Figure 1.4), the first stored-program computer, built at the University of Cambridge by Maurice Wilkes in 1949.

[3] Electronic Delay Storage Automatic Calculator

In von Neumann architecture, the program and data are stored in a single main memory (Figure 1.5), based on the principle that instructions and data are merely sequences of bits with no inherent difference.

The differentiation between the two is made through the access mechanism and context; the program counter accesses the instructions, while the effective address register accesses the data.

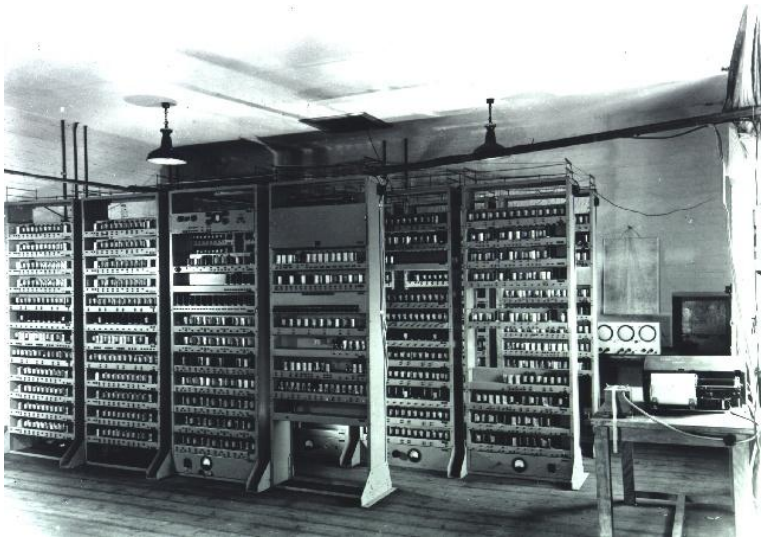


Figure 1.4 : EDSAC

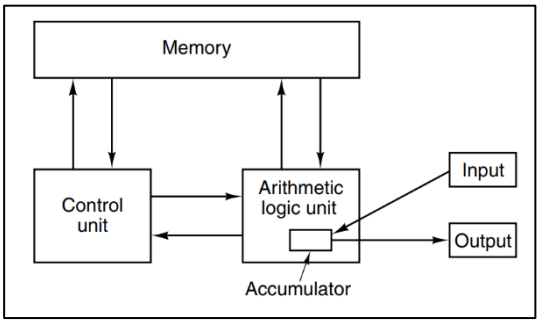


Figure 1.5 : Original von Neumann machine

1.2.2.1 Key Components of von Neumann Architecture

The von Neumann architecture is mainly composed of the following parts:

- **Main memory**: in which data and instructions are stored.
  Memory contains not only the program being executed, but also the operating system, compiler, interpreter, etc.
- **Central Processing Unit (CPU):** reads instructions and data and processes the data then returns the result to memory. CPU is composed of:
  - **Arithmetic and Logic Unit (ALU)**: operates on binary data. It performs arithmetic operations (+, -, *, /…) and logic operations (AND, OR, …).
  - **Control Unit:** searches and interprets instructions and commands the ALU to execute them.
- **Input/Output devices**: allow communication with the outside world. They are used to enter data and instructions and to return the results.

The different parts are interconnected using **system buses.**

1.2.3 Difference between von Neumann architecture and Harvard architecture

The table below shows a comparison between von Neumann and Harvard architectures:

Table 1.1: Comparison between von Neumann and Harvard architectures

von Neumann architecture	**Harvard architecture**
Named after mathematician and computer scientist **John von Neumann**.	Named after the "**Harvard Mark I**", an early relay computer developed at Harvard University.
Uses a **single memory** to store both instructions and data.	Requires **two separate memories** for instructions and data.
Its design is **simple**.	Its design is more **complex**.
Requires only **one bus** for instructions and data.	Requires **two separate buses**, one for instructions and one for data.
The processor requires **two clock cycles** to execute an instruction.	The processor can execute an instruction in **a single cycle**.
Low performance compared to the Harvard architecture.	**High performance** can be achieved.
Lower cost.	**Relatively high cost.**

Chapter 2. Main components of a computer

2.1 THE ARITHMETIC AND LOGIC UNIT (ALU) 16
2.1.1 General presentation 16
2.1.2 Number representation 16
2.1.2.1 Unsigned integers 16
2.1.2.2 Signed integers representation 17
2.1.2.3 Real numbers (floating point encoding) 19
2.1.3 Full adder 21
2.1.4 Example of a simple 1-bit ALU 22
2.2 NOTIONS ABOUT BUSES 23
2.2.1 Definitions 23
2.2.2 Bus types 24
2.2.3 Synchronous and asynchronous buses 25
2.2.4 Parallel and serial transfer 25
2.2.5 Full-Duplex and Half-Duplex 26
2.2.6 Bus throughput 26
2.2.7 Some examples of buses 26
2.2.8 Controller 26
2.2.9 Bus Arbitration 27
2.2.10 Bridge 28
2.2.11 Multiplexing 28
2.3 MEMORY HIERARCHY 29
2.4 REGISTERS 30
2.4.1 User-visible registers 30
2.4.2 Control and Status Registers 31
2.5 MEMORY 31
2.5.1 Some memory characteristics 31
2.5.2 Types of Memory 32
2.5.2.1 Random Access Memory (RAM) 32
2.5.3 Memory system design using integrated circuits (ICs) 33
2.5.4 Storing bytes in memory 34
2.6 CACHE MEMORY 35
2.6.1 Definition and usefulness 35
2.6.2 Locality Principle 35
2.6.3 Operating principle 36
2.6.4 Cache Memory Structure 37
2.6.5 Mapping function 37
2.6.5.1 Direct mapping 37
2.6.5.2 Associative mapping 39
2.6.5.3 Set-associative mapping 40
2.6.6 Replacement algorithms 41
2.6.7 Writing/Reading Policy 41
2.6.7.1 Write through 42
2.6.7.2 Write back 42

2.1 The Arithmetic and Logic Unit (ALU)

2.1.1 General presentation

The arithmetic and logic unit is the unit responsible for performing arithmetic operations such as (+, -, /, ×) and logical operations (AND, OR, NOT, XOR). For this purpose, this unit must be composed of different circuits necessary to perform these operations. The ALU and the registers constitute the **operative part** (or **data path**).

The ALU (Figure 2.1) takes as input two operands A and B which are generally found in the CPU registers. It also takes as input, on n lines, the code of the function (F) to be performed, the synchronization signals as well as a possible carry.

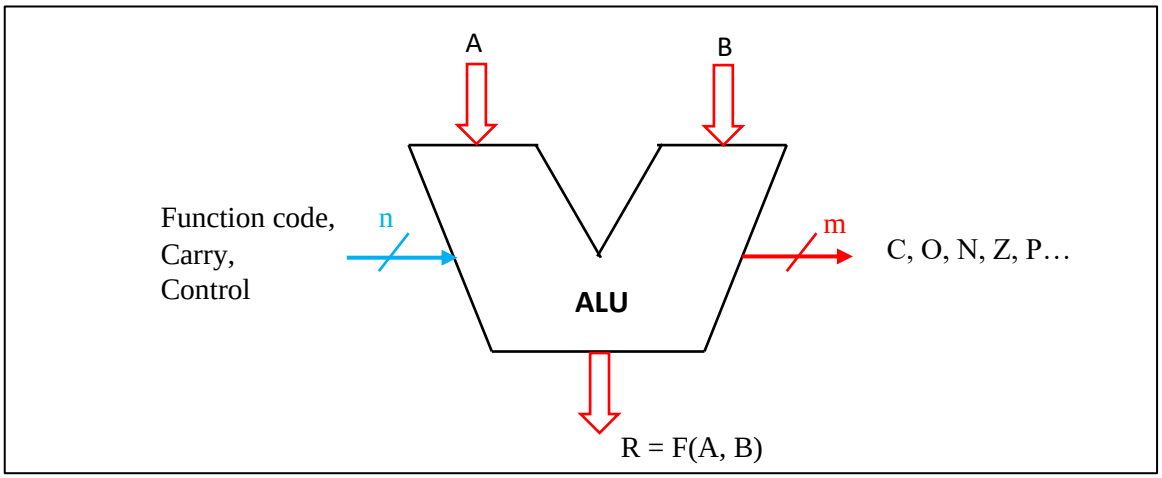
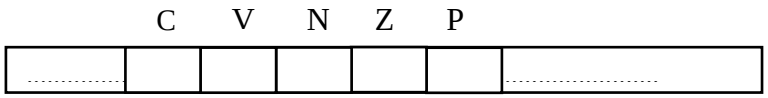


Figure 2.1 Symbolic Representation of ALU

At output, the ALU produces the result R and positions a set of flags (indicators) in the **PSW** (Program Status Word) register.



The main indicators are:

- **C (Carry)**: set to 1 in case of carry.
- **V (oVerflow)**: set to 1 in case of overflow.
- **N (Negative)**: set to 1 in case the result is negative.
- **Z (Zero)**: set to 1 in case the result is zero.
- **P (Parity)**: set to 1 in case the result has even parity.

2.1.2 Number representation

2.1.2.1 Unsigned integers

On n bits, we can represent $\mathbf{2^n}$ unsigned integers (positive or zero). These numbers are between **0** and $\mathbf{2^n-1}$.

For example, on 4 bits, we can represent numbers from 0 to 15 (Table 1.1).

Table 2.1: Representations of unsigned numbers (4-bit example)

Decimal	Binary (Unsigned)
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

2.1.2.2 Signed integers representation

There are several ways to represent signed numbers:

2.1.2.2.1 Sign-absolute value

The most trivial idea to represent signed numbers is to use one bit (the most significant bit) to represent the sign (0 for positive and 1 for negative).

In this case the number of signed integers that can be represented on ***n*** bits are between **$-(2^{n-1}-1)$** and **$2^{n-1}-1$**.

On 4 bits, we can represent numbers between **-7** and **7** (Table 2.2).

Table 2.2: Different representations of signed integers (4-bit example)

Integer	sign-absolute value	1's Complement	2's Complement
7	**0**111	**0**111	**0**111
6	**0**110	**0**110	**0**110
5	**0**101	**0**101	**0**101
4	**0**100	**0**100	**0**100
3	**0**011	**0**011	**0**011
2	**0**010	**0**010	**0**010
1	**0**001	**0**001	**0**001
0	**0**000	**0**000	**0**000
-0	**1**000	**1**111	**0**000
-1	**1**001	**1**110	**1**111
-2	**1**010	**1**101	**1**110
-3	**1**011	**1**100	**1**101
-4	**1**100	**1**011	**1**100
-5	**1**101	**1**010	**1**011
-6	**1**110	**1**001	**1**010
-7	**1**111	**1**000	**1**001
-8			**1**000

Let the number 6 be equal to $(\mathbf{0}110)_2$ in binary. So, -6 is $(\mathbf{1}110)_2$.

Among the disadvantages of this method is that it represents zero in two different ways (+0 and -0).

2.1.2.2.2 One's complement (logical complement)

In this representation, positive integers have the same form as in sign-absolute value, however, negative numbers are obtained by replacing 1s with 0s and 0s with 1s from positive numbers.

On n bits, the signed integers that can be represented are between $\mathbf{-(2^{n-1}-1)}$ and $\mathbf{2^{n-1}-1}$.

By taking the number 6, it keeps the same representation $(0110)_2$. But -6 becomes $(1001)_2$ in 1's complement.

2.1.2.2.3 Two's complement (arithmetic complement)

The 2's complement of a number is obtained by adding 1 to the 1's complement (see the 4-bit example in Table 2.2).

Formally: Comp2(X) = Comp1(X) + 1;

As an example, the positive number 6 keeps the same representation in binary $(0110)_2$. However, to represent (-6) in 2's complement, we proceed as follows:

1. Comp1 (6) = **1**001
2. Let's add the 1: **1**001 +1 = **1010**.

Thus, -6 in 2's complement is represented by $(\mathbf{1010})_2$.

On n bits, the signed integers that can be represented by the 2's complement are between $\mathbf{-2^{n-1}}$ and $\mathbf{2^{n-1}-1}$.

2's Complement Properties

- Comp2 (X) + X = 0
- Comp2 (Comp2 (X)) = X

Addition and subtraction in 2's complement

Addition of two numbers represented in 2's complement is performed as if both numbers were treated as unsigned integers. The result of adding two positive numbers is a positive number and the result of

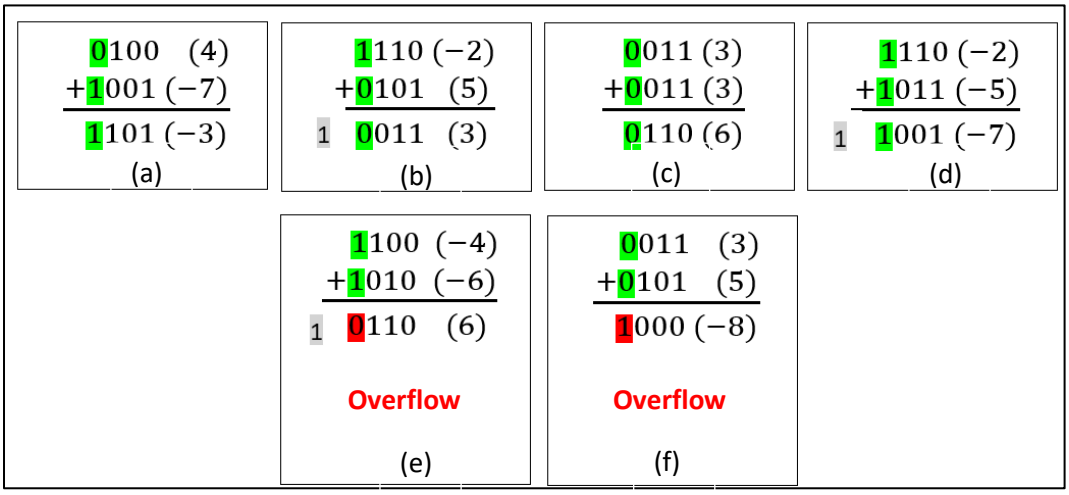


Figure 2.2 : Some examples of the 2's complement addition operation. (a-d): Correct sums, (e, f): Incorrect sums (overflow). The final carry (the 1 in gray) can be ignored (b, d, e).

adding two negative numbers is a negative number. However, if the addition of two numbers with the same sign produces a result with the opposite sign, this indicates an **overflow**. (Examples *e* and *f,* in Figure 2.2)

Subtraction of two numbers (A - B) in 2's complement can be performed by adding the 2's complement of the subtrahend (B) to the minuend (A). i.e., A + (-B). Figure 2.3 illustrates some examples.

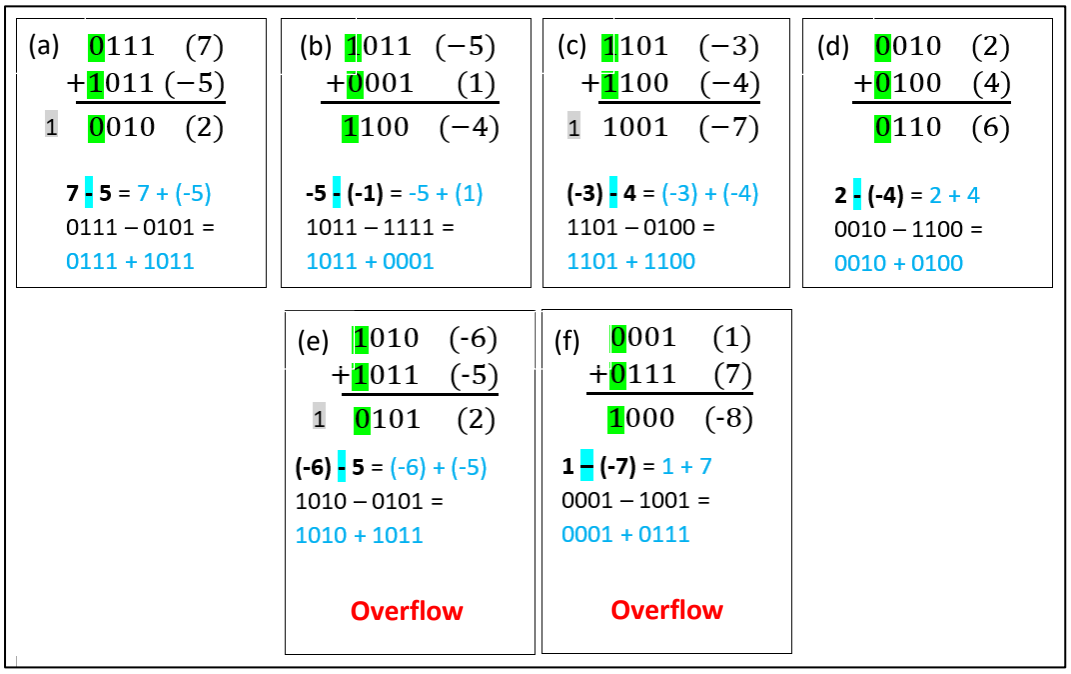


Figure 2.3 : Some examples of the 2's complement subtraction operation (Note the conversion from subtraction to addition). (a-d): correct results, (e, f): incorrect results (overflow). The final carry (the 1 in gray) can be ignored (a, c, e).

2.1.2.3 Real numbers (floating point encoding)

A real number can be represented using floating point in different forms. For example, the number **23.05** can be written in several equivalent forms: **2.305×10^1** or **0.02305×10^3** or **2305×10^{-2}**…etc.

In general, a number **N** in base **b** can be written in the following scientific notation:

$$N = \pm m \times b^e \text{ where } m \in [0, b[\text{ and } e \in \mathbb{Z}$$

Where **m**: is the mantissa and **e**: is the exponent.

Thus, the binary number $(101.11)_2$ can be written as 10.111×2^1, 0.010111×2^4, 10111×2^{-2}, among others.

However, not all representations in the above example are normalized. The normalized representation is the one that best uses the representation space.

IEEE 754 Standard

In single precision, the IEEE 754 standard uses 32 bits to represent a real number. These 32 bits are distributed as follows: 23 low-order bits are reserved for the mantissa, the next 8 bits for the exponent, and the last, high-order bit is used to represent the sign (0 for a positive number, 1 for a negative number). (Figure 2.4)

In double precision, the IEEE 754 standard uses 64 bits to represent a real number. These 64 bits are distributed as follows: 52 low-order bits are reserved for the mantissa, the next 11 bits for the exponent, and the last high-order bit is dedicated to the sign.

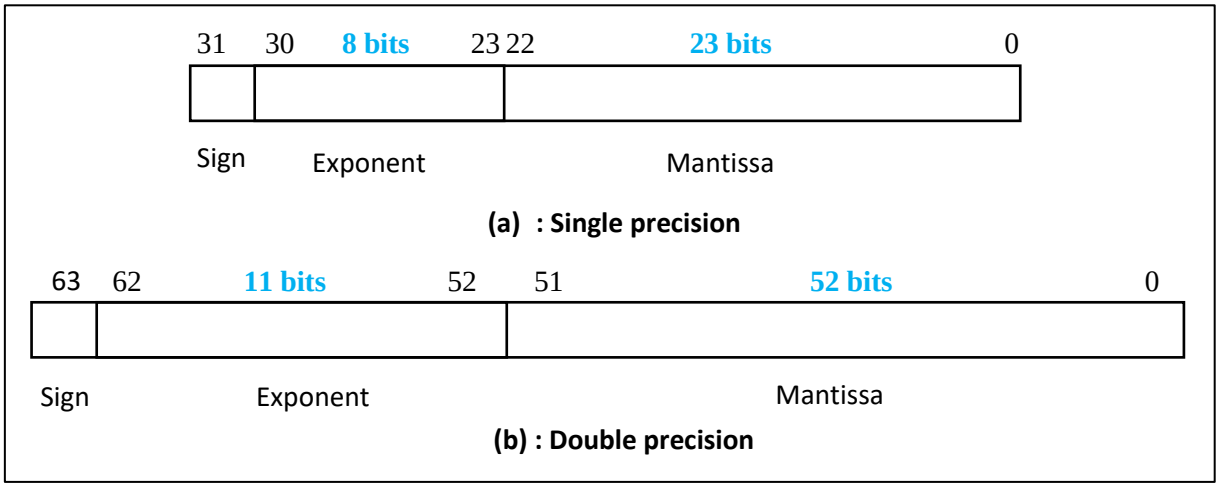


Figure 2.4: The different fields and their sizes of the IEEE 754 standard. (a): Single precision, (b): Double precision

According to this standard, a binary number N is written as follows:

$$N = \pm 1.m \times 2^{E}$$

Where **E = e + excess**; "e" is the exponent in decimal, the **excess** is equal to 127 for single precision and 1023 for double precision.

The bit preceding the decimal point is not coded, being always 1, called the hidden bit.

For example, the normalized representation of the number 5.75 = $(101.11)_2$ in single precision is **1.0111× 2^{129}**. In this representation:

- The sign is + (therefore coded by 0), because the number is positive.
- The mantissa **m** is **0111**.
- The exponent **E** is calculated as follows: **E = e + excess**. In this example, **e**= 2 and the excess is 127, hence **E**=129. Which is written in binary as follows $(10000001)_2$

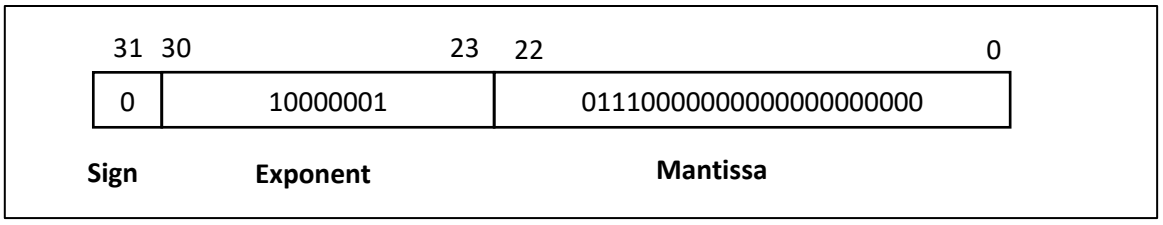


Figure 2.5 : Representation of the binary number (101.11) according to IEEE 754 standard

Table 2.3 gives the representation of particular values according to the IEEE 754 standard. "MAX" value of the exponent means that all bits of this field are set to 1.

Table 2.3: Representation of particular values in IEEE 754

EXPONENT	MANTISSA	VALUE
MAX	0	±∞
MAX	≠0	NaN
0	0	0
0	≠0	Denormalized number

Exercise 2.1:

Let the real number be 9.563, give the representation according to the IEEE 754 standard in single precision.

Solution:

9.563

- Writing the integer part in binary:
  $9 = (1001)_2$
- Writing the fractional part in binary:
  $0.563 = (0.10010000\ldots)_2$
- Normalization:
  $1001.10010000 = 1.00110010000 \times 2^3$ so the exponent e=3
- Internal representation of the exponent:
  $E = e + 127$ therefore $E = 3 + 127 = 130 = (10000010)_2$
- The number is positive so the sign bit is 0
- Representation of the number sign-exponent-mantissa according to the IEEE 754 standard (single precision) sign-exponent-mantissa:

0 || 10000010 || 00110010000001000001100001000000000000000

0	10000010	00110010000001000001100
Signe	**Exposant**	**Mantisse**

2.1.3 Full adder

A full adder is an adder that takes into account, in addition to the two operands (one bit for each), the previous carry, and generates as output a result bit and another carry bit.

c_{in}	a	b	s	c_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

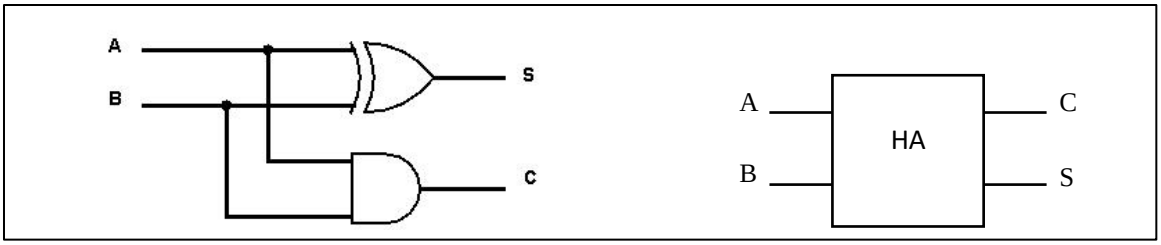


Figure 2.6: Half adder and its symbolic representation

A full adder can be made using two half adders as follows:

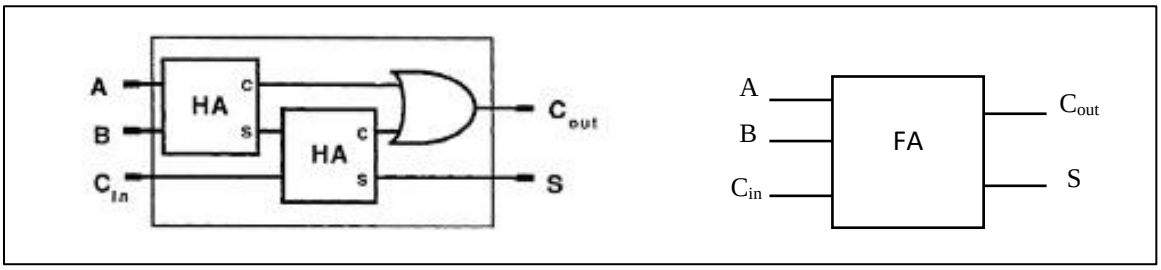


Figure 2.7: Full adder and its symbolic representation

Finally, to realize an **n-bit adder**, we use **n full adders**. The carry-out of an adder of stage **i** is used by the adder of stage **i+1** as the carry-in.

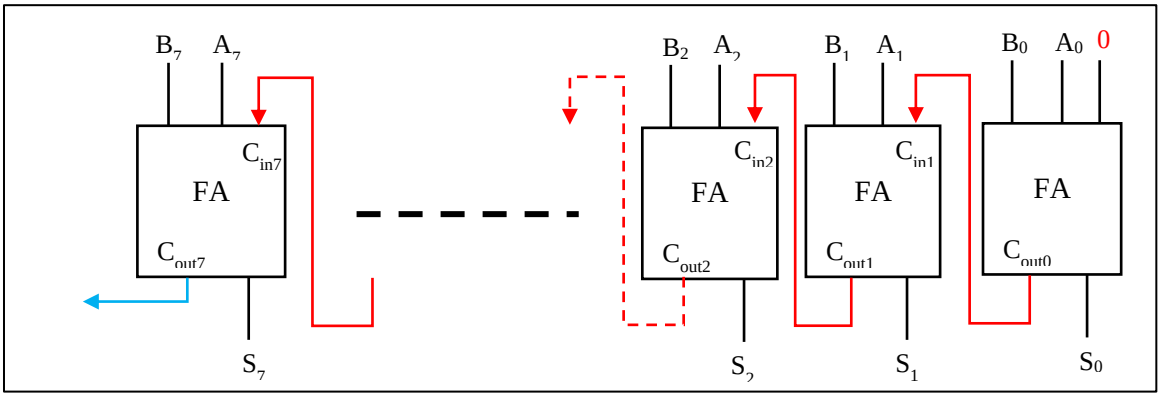


Figure 2.8: Example of eight-bit ripple-carry adder

2.1.4 Example of a simple 1-bit ALU

Figure 2.9 below illustrates a simple 1-bit ALU. This unit is capable of performing 4 operations; 3 logical operations ($A \cdot B$, $A + B$, $\bar{B}$) and one arithmetic operation (Addition).

The decoder has two inputs (F_0 et F_1) and therefore 4 outputs. The decoder allows selecting one operation among the four.

ENA and **ENB** allow the activation of inputs A and B respectively.

INVA allows inverting input A, it is useful to perform subtraction.

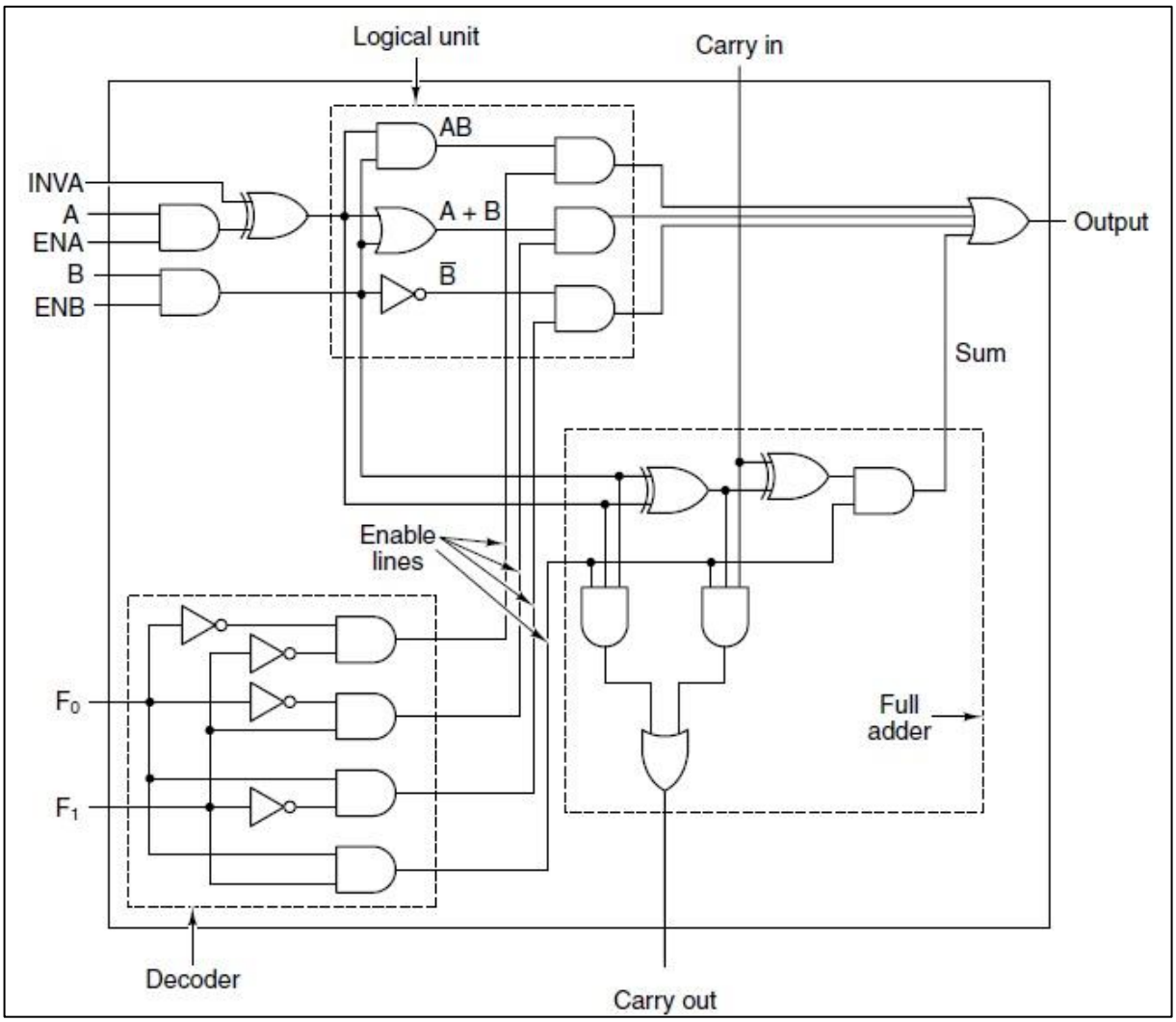


Figure 2.9: A single-bit Arithmetic Logic Unit [1]

2.2 Notions about buses

2.2.1 Definitions

- *"A bus is the digital communication mechanism used within a computer system to interconnect functional units."* [13]
- *"A bus is a shared communication link, which uses one set of wires to connect multiple subsystems."* [9]
- *"A bus in computer terminology represents a physical connection used to carry a signal from one point to another."* [12]
- *"A bus is simply a set of **n** conductive wires, used to carry **n** binary signals."* [4]

[4] Translated: « *Un bus est simplement un ensemble de n fils conducteurs, utilisés pour transporter n signaux binaires.* » [25]

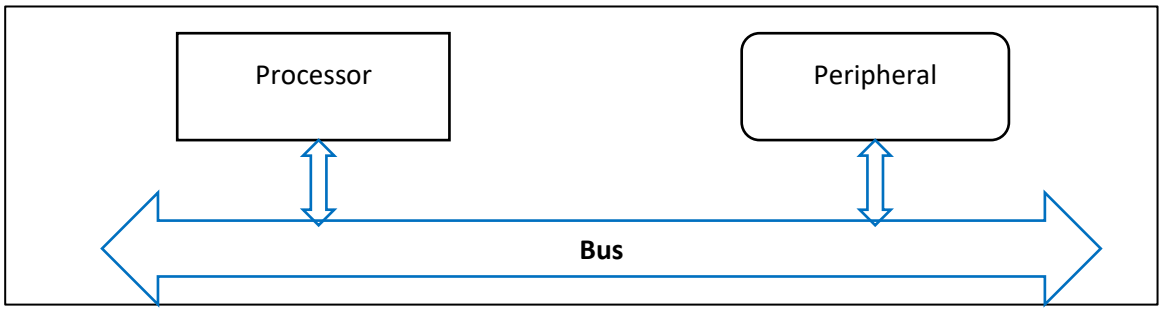


Figure 2.10: Illustrative image of a bus

2.2.2 Bus types

Depending on the type of information transported, three types of buses are distinguished (Figure 2.11):

1. **Address Bus**: it is a unidirectional bus allowing the processor to convey addresses to the peripherals.
   The lines of this bus allow the selection of a memory location (for reading/writing) or to designate a peripheral. Thus, the width of this bus gives an idea on the address space that the processor can address.
   *Note: An address bus of width **n** lines allows **2^n** memory words to be addressed.*
2. **Data Bus**: It is a bidirectional bus that allows data to be transported from the processor to the peripherals and vice versa.
   The width of the data bus gives an idea of the amount of data transmitted at the same time.
3. **Control Bus:** It is a bidirectional bus that carries control signals that enable access control and use of the address and data buses.
   Control signals specify the operations to be performed. Typical control lines include:
   - ***Memory write***: causes data on the bus to be written into the addressed location.
   - ***Memory read***: places the data from the addressed location on the bus.
   - ***I/O write***: causes data on the bus to be sent to the addressed I/O port.
   - ***I/O read***: allows placing the data from the addressed I/O port on the bus.
   - ***Transfer ACK***: indicates that data have been accepted from, or placed on the bus.
   - ***Bus request***: indicates that a module needs to take control of the bus.
   - **Bus grant**: indicates that a requesting module has been granted control of the bus.
   - ***Interrupt request***: indicates that an interrupt is pending.
   - ***Interrupt ACK***: Acknowledges that the pending interrupt has been recognized.
   - ***Clock***: used to synchronize operations.
   - ***Reset***: initializes all modules.

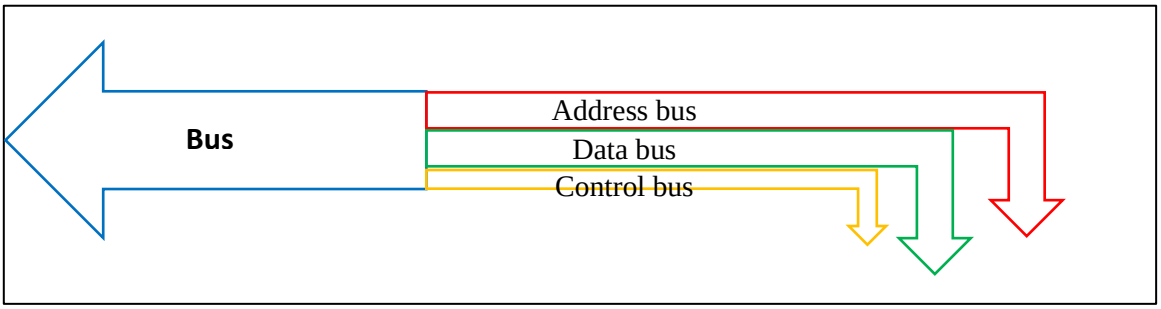


Figure 2.11: Bus types

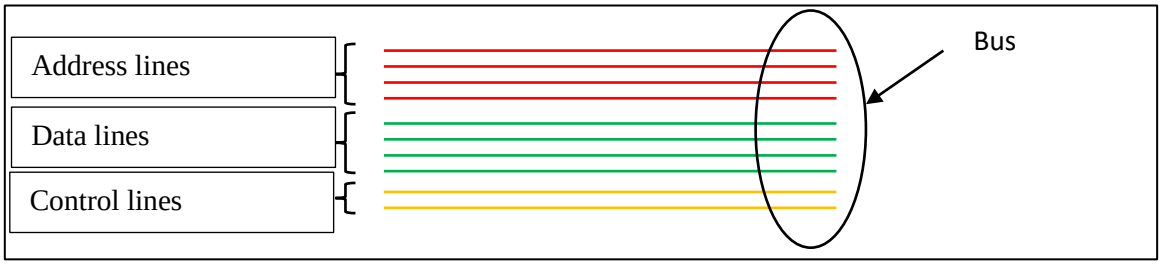


Figure 2.12: Conceptual division of wires that comprise a bus into lines for control, addresses, and data

2.2.3 Synchronous and asynchronous buses

Data reception and transmission can occur at specific times determined by timing signals. As a result, a bus can be either synchronous or asynchronous.

- **Synchronous bus**: If data transfer on the bus is controlled by a bus clock, this serves as a time reference for all signals on the bus.
- **Asynchronous bus**: if data transfer depends on data availability rather than a clock signal. Data transfer on an asynchronous bus uses a technique called "handshaking".

2.2.4 Parallel and serial transfer

- **Parallel transfer** consists of transferring several bits simultaneously. This is only possible if the bus is composed of several lines. (Figure 2.13)
- **Serial transfer** only allows the transfer of a single bit at a time. The main advantage of this transfer mode lies in the small number of wires used. (Figure 2.13)

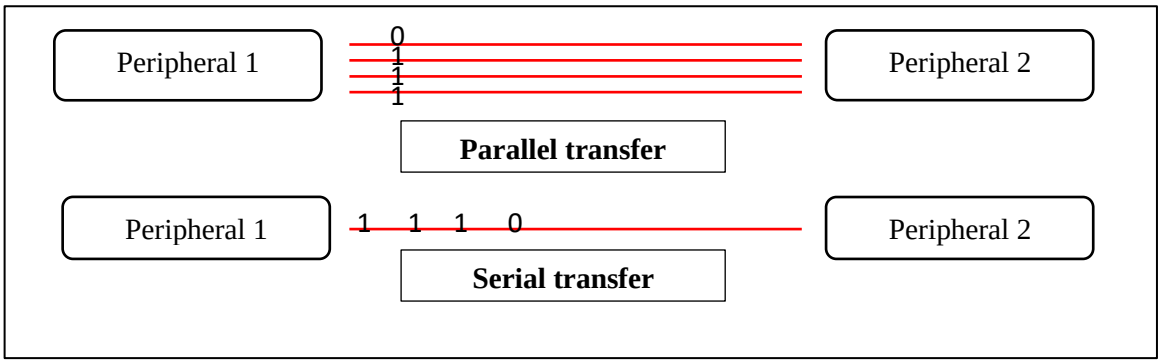


Figure 2.13: Parallel and serial transfer

2.2.5 Full-Duplex and Half-Duplex

A controller that allows data to be transferred simultaneously in both directions is known as **Full-Duplex**. When a controller allows data to be transferred in only one direction at a time, it is said to be **Half-Duplex**. (Figure 2.14)

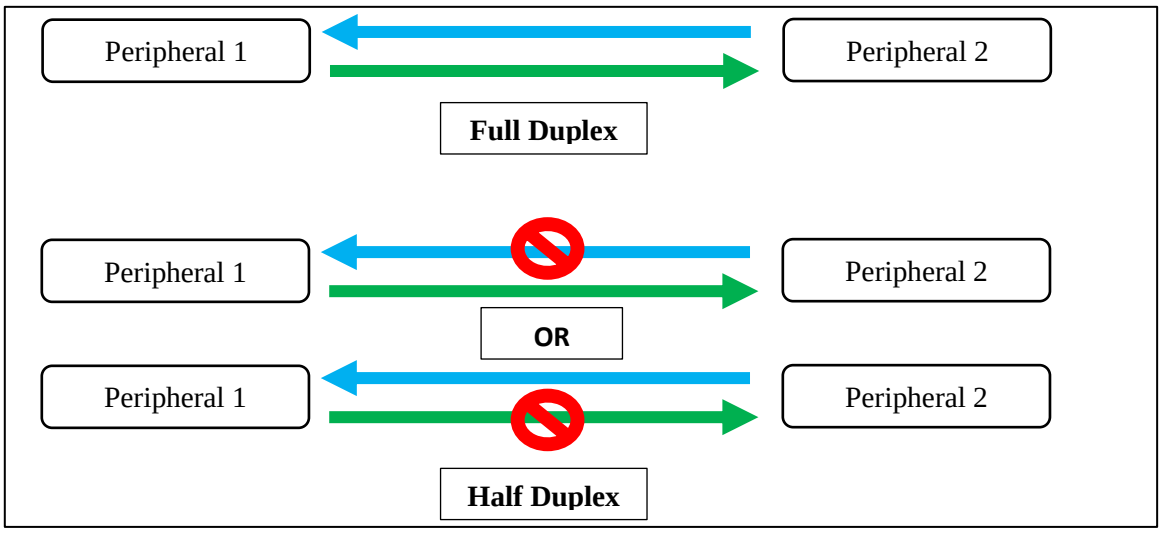


Figure 2.14: Full-duplex and half-duplex transmission

2.2.6 Bus throughput

The throughput (or bandwidth or transfer rate) depends on the electrical characteristics, the method used to perform the transfers (synchronous and asynchronous modes) and the bus exchange protocol.

The maximum throughput of a bus is the product of its width and its frequency. Throughput is measured in **bytes/s**.

$$Bus\ throughput\ =\ width\ \times\ frequency$$

2.2.7 Some examples of buses

The table below illustrates the characteristics of some buses.

Table 2.4: characteristics of some buses

BUS	width	frequency	throughput	Use
ISA (Industry Standard Architecture)	16 bits	8.33 MHz	16.7 MB/s	
PCI (Peripheral Component Interconnect)	64 bits	66 MHz	528 MB/s	Processor/non-graphics device
AGP (Accelerated Graphics Port)	32 bits	66 MHz x 8	2.1 GB/s	Processor/Graphics Card
SCSI (Small Computer System Interface)	16 bits	40 MHz	80 MB/s	Exchanges between devices

2.2.8 Controller

Each I/O device is composed of two components: a controller, which contains most of the electronic circuitry, and the physical I/O device itself, such as a disk drive (Figure 2.15). The function of the controller is to control its I/O device and handle bus access for it.

A controller is said to perform direct memory access when it reads or writes from or to memory without processor intervention. This mechanism is known by the acronym **DMA** (**Direct Memory Access**).

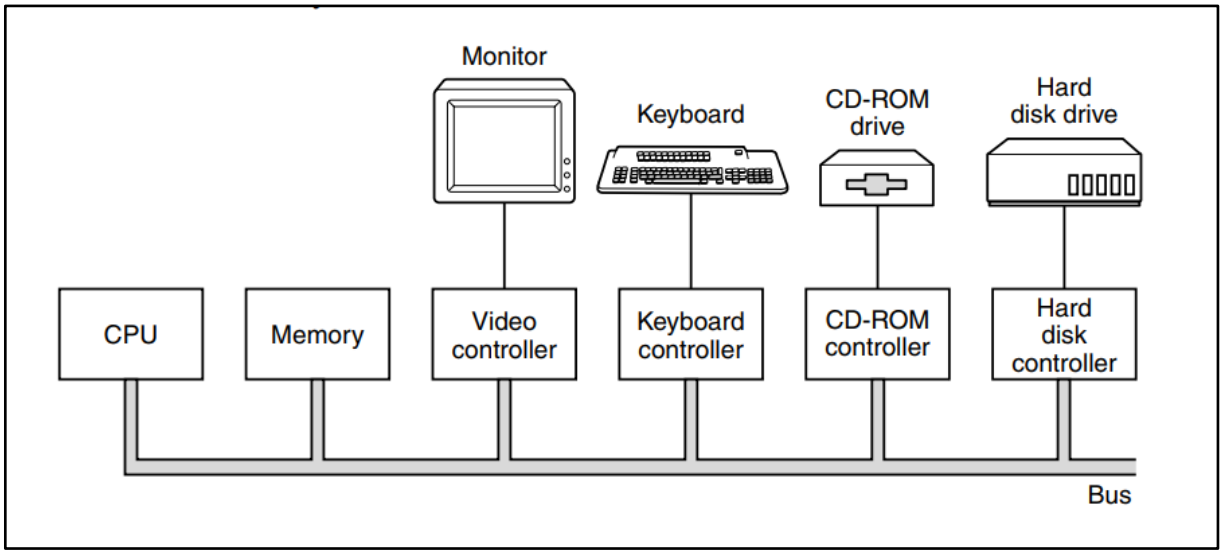


Figure 2.15: Logical structure of a simple personal computer [1]

2.2.9 Bus Arbitration

When multiple devices are connected to a single bus, several of them may request access to the bus at the same time. However, if multiple devices access the bus there will be a conflict. For this, it is necessary to have a mechanism that allows managing (arbitrate) the access to the bus.

Bus arbitration is "The process of determining which competing bus master will be permitted access to the bus."

The bus arbitration can be centralized or decentralized:

- **Centralized arbitration**: one of the components is designated as an arbiter. The others make a request for access to the bus via a dedicated line of the control bus and the arbiter grants the request of one of them according to priorities established in advance.
  The disadvantage of this type of arbitration is the overload of the arbiter.
  One solution is to add a chip, referred to as "bus arbiter" that allows deciding which peripheral will have access to the bus.
- **Decentralized arbitration**: No arbiter is designated, but the different elements discuss among themselves to allocate the bus. When one of them wants to obtain it, it sends a request on special lines of the control bus and compares its priority level with that of the other elements that have also sent an access request. The highest priority then knows that the bus is allocated to it and that it can occupy it.

2.2.10 Bridge

Bridges allow the interconnection of multiple heterogeneous environments. In fact, with a bridge we can interconnect two types of buses. This allows to take advantage of peripherals compatible with each of the buses on the same computer. (Figure 2.16).

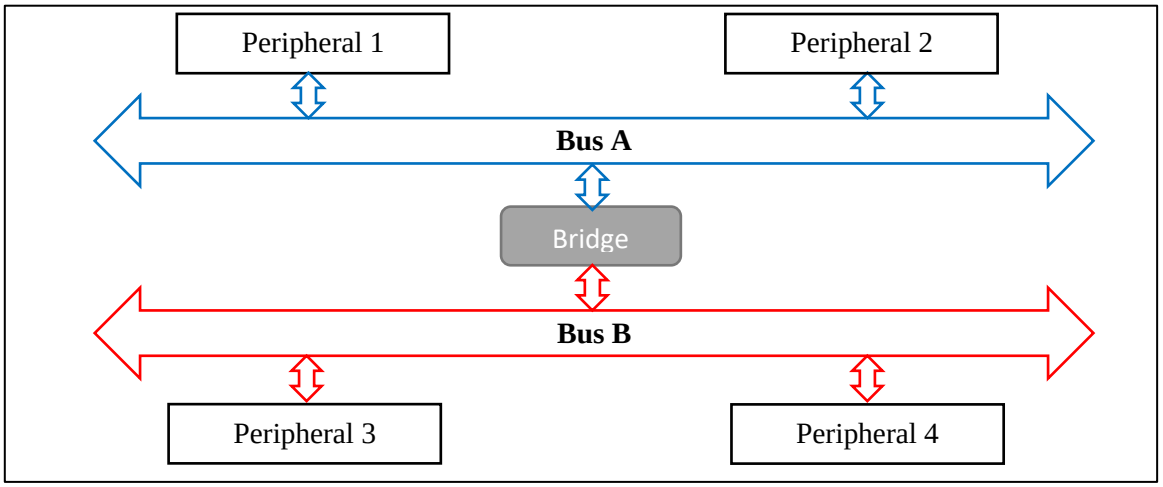


Figure 2.16: Illustration of a bridge connecting two buses

Figure 2.17 shows the architecture of an old Pentium system. This system has several types of buses, such as the memory bus, PCI bus, and ISA bus, with different bandwidths. Two bridges are used to connect these different buses; the PCI bridge connects the processor, main memory, and the PCI bus. As for the ISA bridge, it connects the PCI bus to the ISA bus and also supports one or two IDE disks.

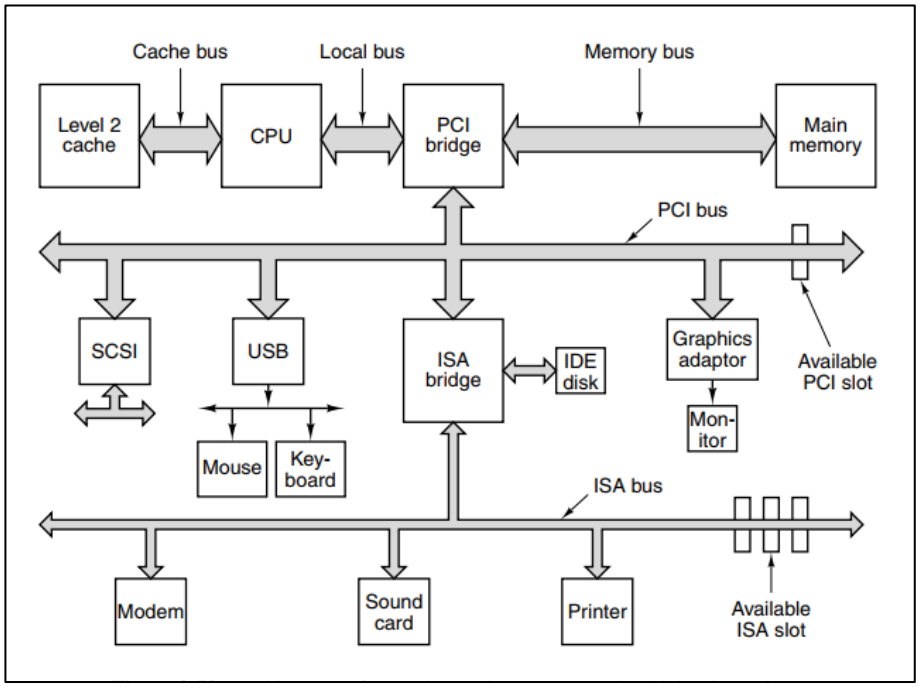


Figure 2.17: Architecture of an early Pentium system. [1]

Note that thicker buses indicate greater bandwidth than thinner ones (not to scale).

2.2.11 Multiplexing

Although increasing the bus width improves the throughput, it leads to the occupation of more space as well as more electronic components. So, the solution is the use of **multiplexing**.

Data multiplexing involves dividing large data into smaller pieces of the same size as the bus and passing them over the bus one by one (Figure 2.18). Multiplexing is performed by a circuit called a **multiplexer**.

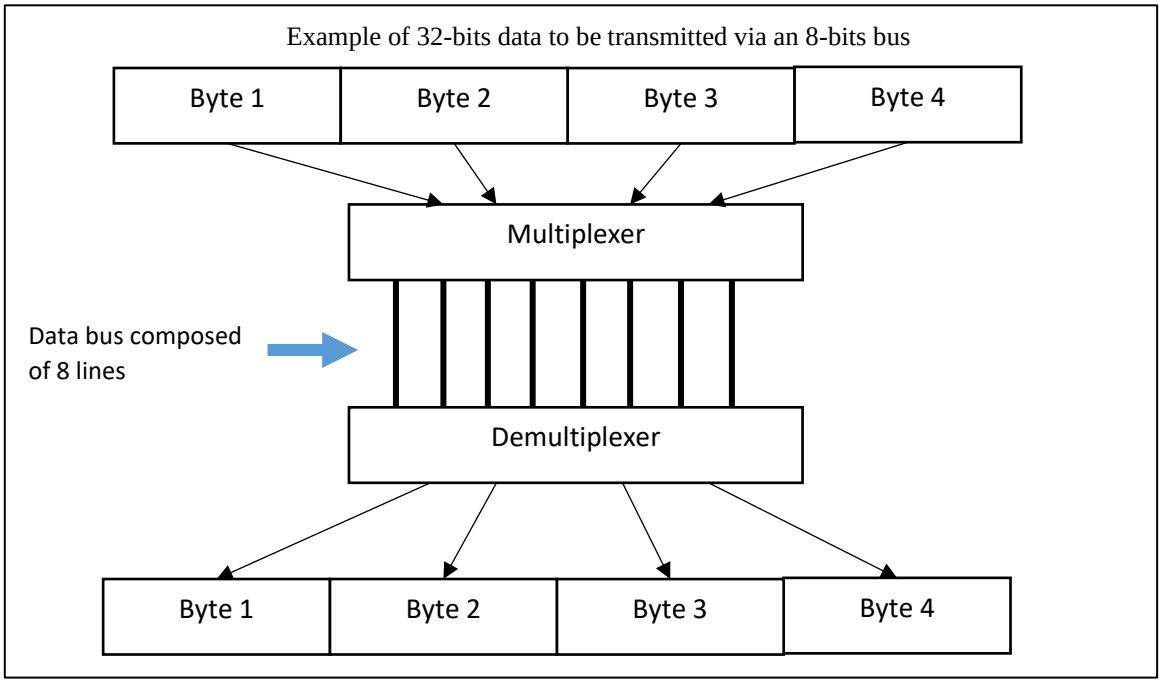


Figure 2.18: Data multiplexing

2.3 Memory hierarchy

Computer memory is organized in a hierarchy (Figure 2.19), the faster the memory, the lower its capacity and vice versa. As for the cost per bit, it decreases from the fastest level to the slowest level.

The memories closest to the processor are the registers (in the CPU). These are the fastest of all types of memories. In the next level, we find the cache memory which is intended to speed up access to the main memory by storing the most used data. The main memory is intended for storing programs currently running and their data. The main memory is followed by the secondary memory (for example hard disks) and the archiving memories (CD-ROM, DVD-ROM, etc.) which are distinguished by their large capacities, but they have very long access times compared to the memories of the previous levels.

The goal behind designing a memory hierarchy is to have a memory system that operates as if it consists entirely of the fastest unit and whose cost is dominated by the cost of the slowest unit.

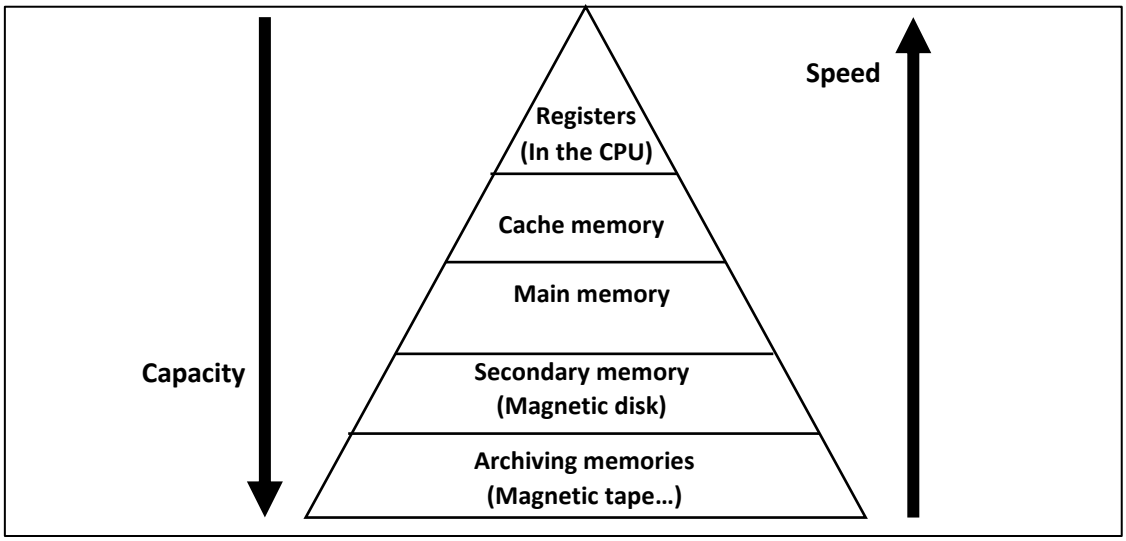


Figure 2.19: Memory hierarchy

2.4 Registers

Registers are storage areas located inside the processor. They are of low capacity, however, with a very fast access time. A register is used to temporarily store data, an instruction, or an address. Generally, their size is a power of two (8, 16, 32 or 64 bits), and their number depends on the processor architecture and typically varies between ten and a hundred.

A register can be either general-purpose or special-purpose. **General-purpose registers** can be used for any task and can hold different types of information: data, addresses, etc., at different times. In contrast, **special-purpose registers** have specific functions within the processor and can only hold a particular type of information.

The processor registers have two roles:

- ✓ ***Optimizing memory access***: The so-called **user-visible registers** allow the programmer, in machine language or in assembly, to reduce access to the main memory by using these registers to temporarily store and manipulate data. This simplifies data management and speeds up access to information.
- ✓ ***Processor control and management***: The **control and status registers** are used by the processor control unit as well as by the privileged programs of the operating system to manage the overall operation of the processor and supervise the execution of programs.

2.4.1 User-visible registers

User-visible registers are those registers that are directly accessible by the programmer using the assembly language or machine language that the processor is executing. These registers are generally divided into several categories:

- ✓ **General purpose registers**: These registers can be assigned to various functions by the programmer. Any general-purpose register can hold the operand for any opcode. In some cases, general purpose registers can be used for addressing functions.

- ✓ **Data registers**: Data registers are used exclusively to hold data and cannot be used in calculating an operand address.
- ✓ **Address registers**: These registers hold memory addresses and are often used to point to data in memory. Among the address registers:
    - **Stack Pointer (SP)**: is a special register of the processor. It provides the address of a memory word called the top of the stack. The top of the stack usually has the same address as the stack word with the smallest address n. Stack pointers are mainly used in the function or subroutine calling mechanism.

The stack is a segment of memory that stores contextual information related to the currently executing program. When a procedure is called, a portion of the stack is usually allocated to store the procedure's local variables, the return address after its execution, and other control information.

    - **Segment pointers**: In a segmented addressing machine, a segment register contains the address of the base of the segment. There can be multiple registers: for example, one for the operating system and one for the current process.
    - **Index registers**: These are used for indexed addressing and can be self-indexing.
- ✓ **Flags register**: A register that is partially visible to the user and contains flags: a set of bits that are automatically updated by the processor hardware after each operation and the programmer cannot change them. These bits are called **flags** and are useful when executing conditional branch instructions. Typically, these flags are part of the **PSW** (**Program Status Word**) register that contains status information.

2.4.2 Control and Status Registers

Various processor registers are used to manage and control its operation. In most architectures, these registers are not visible to the user. However, some may be accessible to machine instructions executed in control mode or operating system mode.

- ✓ **Program Counter (PC)**: Contains the address of an instruction to be retrieved.
- ✓ **Instruction Register (IR):** Contains the last instruction retrieved.
- ✓ **Memory Address Register (MAR)**: Contains the address of a location in memory.
- ✓ **Memory Buffer Register (MBR)**: Contains a word of data to be written to memory or the last word read.

2.5 Memory

Memory is one of the main components of the computer. Its function is to memorize information whether programs or data. It allows this information to be recorded (written) and retrieved later (read). In fact, a computer has a set of memories of different types, which can be internal memories or external memories. Internal memory is the memory that is directly accessible by the processor, includes registers, cache memory, and central memory. External memory, on the other hand, is composed of peripheral storage devices that are accessible to the processor through input/output (I/O) controllers. Hard drives, for example, are part of this type of memory.

2.5.1 Some memory characteristics

The memory of a computer system has several key characteristics that determine its performance and efficiency. Among others, we cite the following characteristics:

- ✓ **Capacity**: is the maximum number of units (bits, bytes, words) of data that the memory can store.
- ✓ **Access time**: time elapsed between the start of a read/write operation and its completion.
- ✓ **Bandwidth**: This criterion is expressed as the product of the width of the data bus and the frequency of the memory.
- ✓ **Memory cycle**: is the minimum time between two successive accesses to the memory (Cycle > access time). Because additional operations are necessary between two accesses (signal stabilization, synchronization, etc.)

2.5.2 Types of Memory

Depending on the mechanism used to store or retrieve data, a memory can be classified into one of the following four types:

1. **Random Access Memory (RAM)**
   a. **Read/Write Memory (RWM)**
   b. **Read-only memory (ROM)**
2. **Content-addressable memory (CAM)**
3. **Sequential-access memory (SAM)**
4. **Direct-access memory (DAM)**

2.5.2.1 Random Access Memory (RAM)

A memory is organized in several lines. Each line is associated with an address (Figure 2.20). Through an address bus (and address register) of size **n**, $\mathbf{2^n}$ lines can be addressed.

Each memory line is composed of a fixed number of **m** bits, is known as a **Memory Word**. Generally, data transfer is done by memory word.

Therefore, the size of the memory in bits is $\boldsymbol{2^n \times m}$ bits.

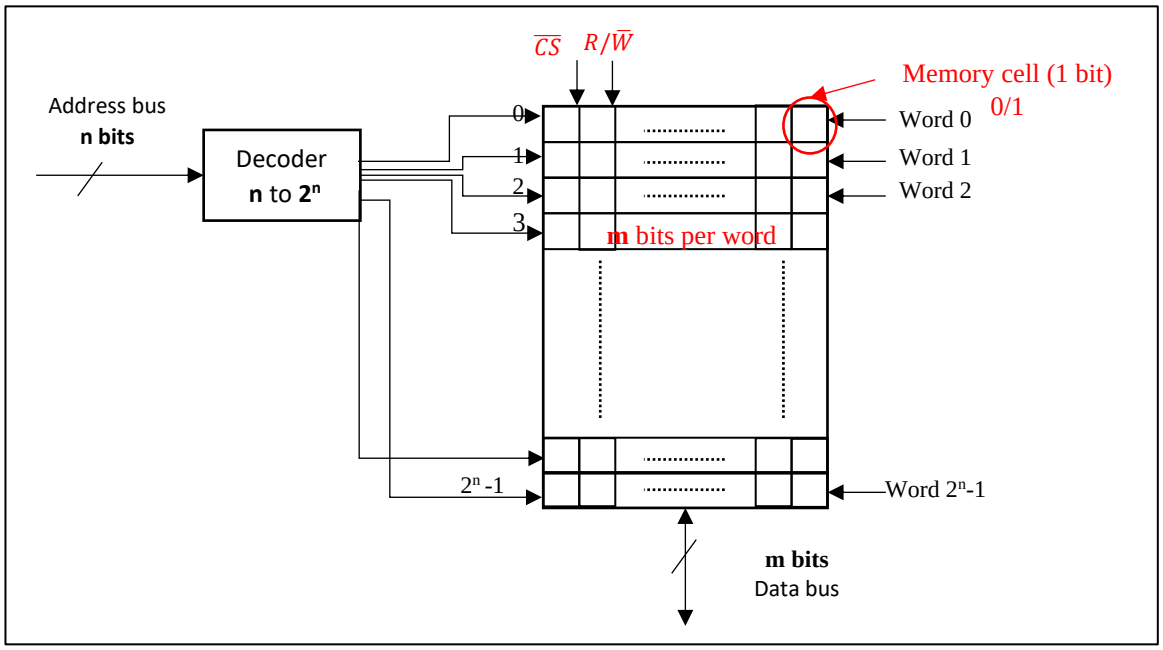


Figure 2.20: General diagram of a memory

Example 01: Consider a RAM of size 2 K words. Knowing that the size of the memory word is 4 bits. Calculate its size in bits and bytes. (1 K= 2^{10})

Size of this memory in bits = 2 K × 4 = 2 × 2^{10} × 2^2 bits = 2^{13} bits.
Size of this memory in bytes= 2^{13} × 2^{-3} = 2^{10} bytes.

Example 02: Consider a RAM of size 4 KB. Knowing that the size of the memory word is 16 bits. Calculate its size in bits and in memory words.

Size of this memory in bits = 2^2 × 2^{10} × 2^3 = 2^{15} bits
Size of this memory in memory words = 2^2 × 2^{10} × 2^{-1} = 2^{11} words

Two types of read/write memory (RWM) (random access memory) are available: static (**SRAM**) and dynamic (**DRAM**).

2.5.2.1.1 SRAM memories

- ✓ In SRAM memories, a memory cell (one bit) is made by a flip-flop (memorizing a state requires 4 to 6 transistors).
- ✓ The information in this type of memory can be stored as long as the power supply is maintained.
- ✓ They have an access time of 5 to 10 nanoseconds, they are very fast. But they have a higher cost and consumption per bit than dynamic memory.
- ✓ They have low capacities because they have a lower integration density per bit than DRAM.
- ✓ They are reserved for the implementation of cache memories.

2.5.2.1.2 DRAM memories

- ✓ In DRAM memories, a memory cell (one bit) is made by a capacitor and a transistor. Therefore, DRAM has an integration density per bit 4 to 6 times greater than SRAM.
- ✓ The major drawback of DRAM is the discharge of the capacitor (in a few milliseconds), so this type of memory is volatile even under power, and therefore requires regular refreshing.
- ✓ They have an access time between 50 and 60 nanoseconds, so they are slow compared to SRAM memories.
- ✓ The cost and consumption per bit of DRAM are low compared to SRAM.
- ✓ DRAM memories are reserved for the implementation of central memories.

2.5.3 Memory system design using integrated circuits (ICs)

It is possible to create a memory system of required size and other characteristics, by combining several memory ICs.

These ICs are combined for:

- Increasing the size of memory words. (Figure 2.21)

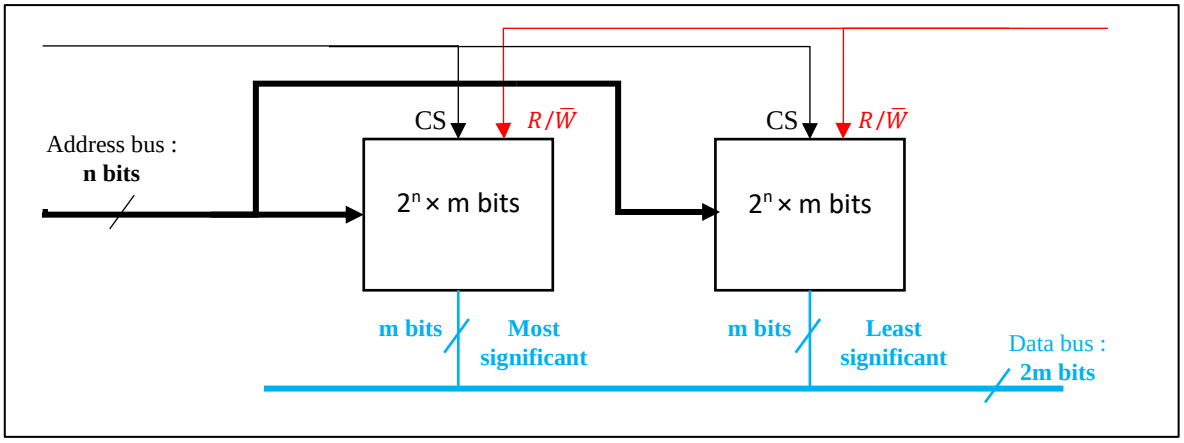


Figure 2.21: Example of increasing word size using 2 ICs.

- Increasing the number of words in memory. (Figure 2.22)

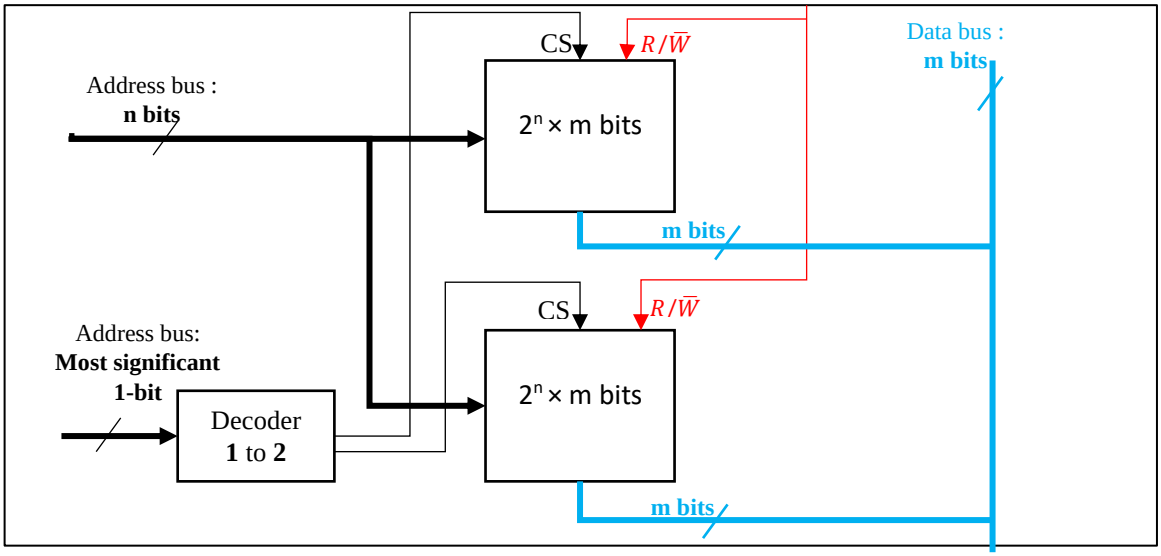


Figure 2.22: Example of increasing the number of words using 2 ICs.

To determine the number of ICs and the arrangement of these ICs

1 Determining the number of ICs, **N**: $N = \frac{Total\ memory\ capacity}{chip\ capacity}$

2 Arrangement of the **N** ICs in a P×Q matrix such that:

$$Q = \frac{number\ of\ bits\ per\ word\ in\ memory\ system}{number\ of\ bits\ per\ word\ in\ the\ IC}$$

And

$$P = \frac{N}{Q}$$

2.5.4 Storing bytes in memory

There are 2 modes to store the bytes of a word in memory:

1. **Little endian**: The least significant byte is stored first (i.e., at the lowest address).
2. **Big endian**: The most significant byte is stored first.

Example: Consider the following value (in Hexadecimal): **0xABCDEF12** (4 bytes)

Address	i	i+1	i+2	i+3
Little endian	12	EF	CD	AB
Big endian	AB	CD	EF	12

2.6 Cache Memory

2.6.1 Definition and usefulness

Cache memory is a small but extremely fast intermediate memory located between the processor and the main memory (Figure 2.23). It plays a crucial role in temporarily storing data frequently used by the processor, thus allowing faster access compared to the main memory. Thanks to its proximity and speed, the cache memory reduces the waiting time for the processor, thus improving the overall performance of the system. In other words, the reading of information by the CPU is accelerated by placing it in the cache in advance.

The cache memory contains a copy of parts (blocks) of memory, which are made up of a fixed number of words.

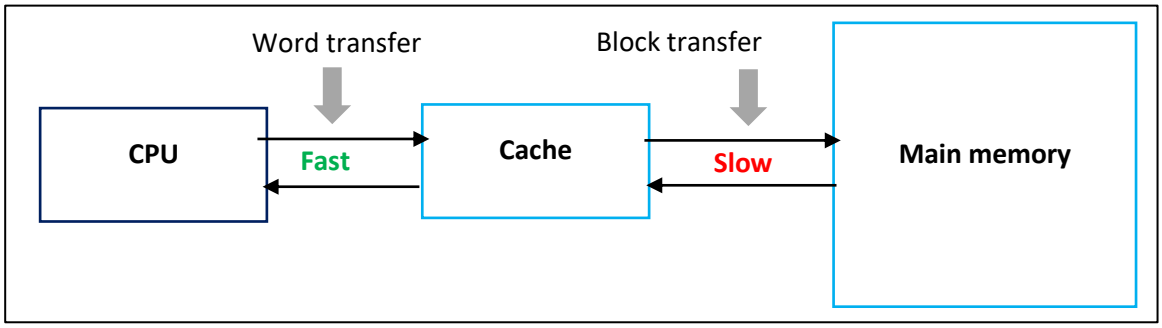


Figure 2.23: Illustrative image of cache and main memory

2.6.2 Locality Principle

Caches exploit two types of address locality to achieve their goal:

1. **Spatial locality**: When the processor accesses a memory address, it is likely that it will also access neighboring addresses shortly after.
2. **Temporal locality**: Memory locations that have been used recently have a high probability of being accessed again in the near future.

2.6.3 Operating principle

Cache memory stores a copy of some data from main memory. When the processor tries to read a word from memory, it first checks whether this word is in the cache. If it is successful (called a **cache hit),** the word is immediately transmitted to the processor, thus speeding up access. On the other hand, if the word is not present in the cache (called a **cache miss**), an entire block of the main memory, containing a fixed number of words, is loaded into the cache, then the required word is transmitted to the processor. The following algorithm summarizes cache read access and cache miss handling:

```
if word is present then // cache hit
      load processor with word;
else // cache miss
      if cache is full then
            load cache(replace);
            load processor;
      else
            load cache;
            load processor;
      endif
endif
```

Average Access Time

The effective time to access data is:

$$T_{eff} = h \times T_c + (1 - h) \times T_m$$

Where T_c denotes the cache access time, and T_m denotes the main memory access time—or, more generally, the mean access time of the lower levels of the memory hierarchy (*Miss time*). The parameter *h* represents the hit rate.

Example: Suppose that 70% of a program's instructions are located in the cache memory, which has an access time of 5 ns. Given that the access time to the main memory is 60 ns, calculate the average memory access time.

The cache hit rate is 70% because 70% of the instructions were found in the cache.
We also have the main memory access time (T_m) is 60 ns and the cache memory access time is 5 ns. Therefore:

$$T_{eff} = h \times T_c + (1 - h) \times T_m$$
$$T_{eff} = 0.7 \times 5 + (1 - 0.7) \times 60$$
$$T_{eff} = 3.5 + 18$$
$$T_{eff} = 21.5$$

So, the effective (average) access time for an instruction is 21.5 ns.

2.6.4 Cache Memory Structure

Cache memory is organized into fixed-size blocks called *cache lines*, each typically consisting of 4 to 64 consecutive bytes. Each cache entry consists of three elements (Figure 2.24):

1. **Validity bit**: This bit indicates whether the entry contains valid data.
2. **Tag**: This field consists of a unique value that identifies the corresponding block of memory from which the data originated.
3. **Data**: This field contains a copy of the data from main memory, corresponding to a cache line.

So, the number of blocks **M** of size **K** words of a main memory containing $\mathbf{2^n}$ memory words is:

$$M = \frac{2^n}{K}$$

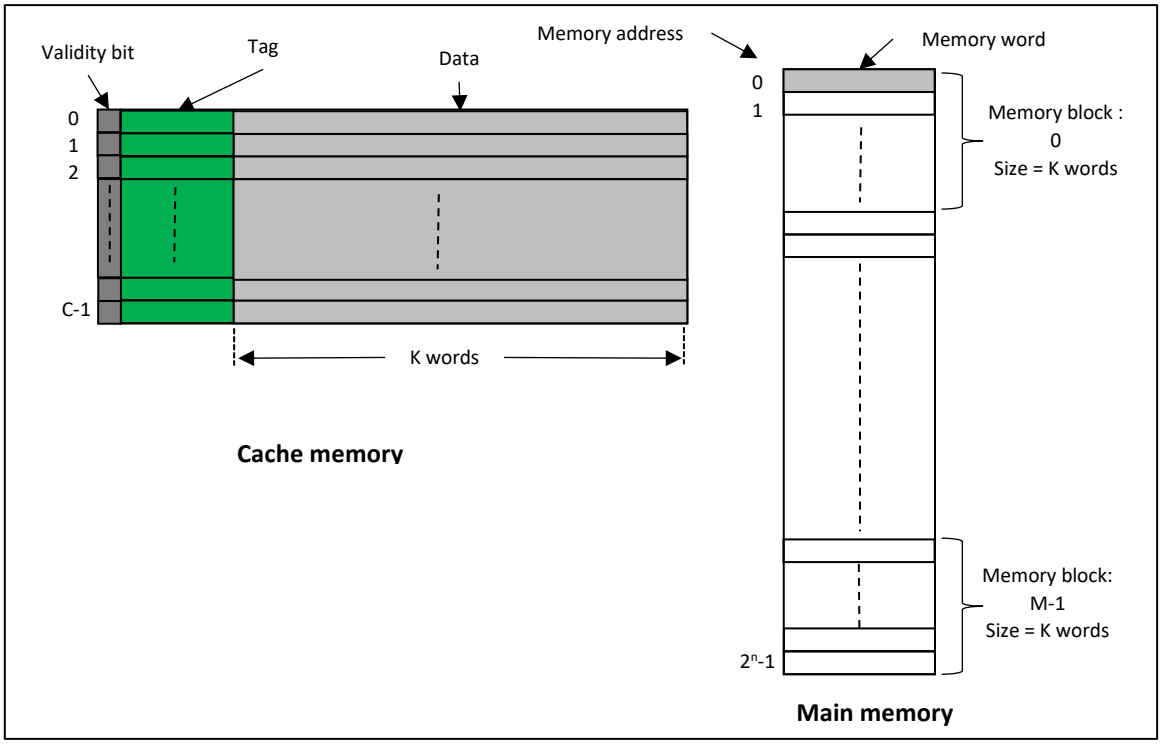


Figure 2.24: Structure of cache and main memory

2.6.5 Mapping function

The number of cache lines **C** is much smaller than the number of blocks in main memory **M** (C<<M). Therefore, a specific cache line cannot be dedicated *only and always* to a particular memory block.

2.6.5.1 Direct mapping

In this type of cache, each block **j** of the main memory has a single reserved line **i** in the cache where it will be stored (Figure 2.25). This storage location **i** is determined as follows:

$$i = j \ modulo \ C$$

Where **C**, is the number of lines in the cache.

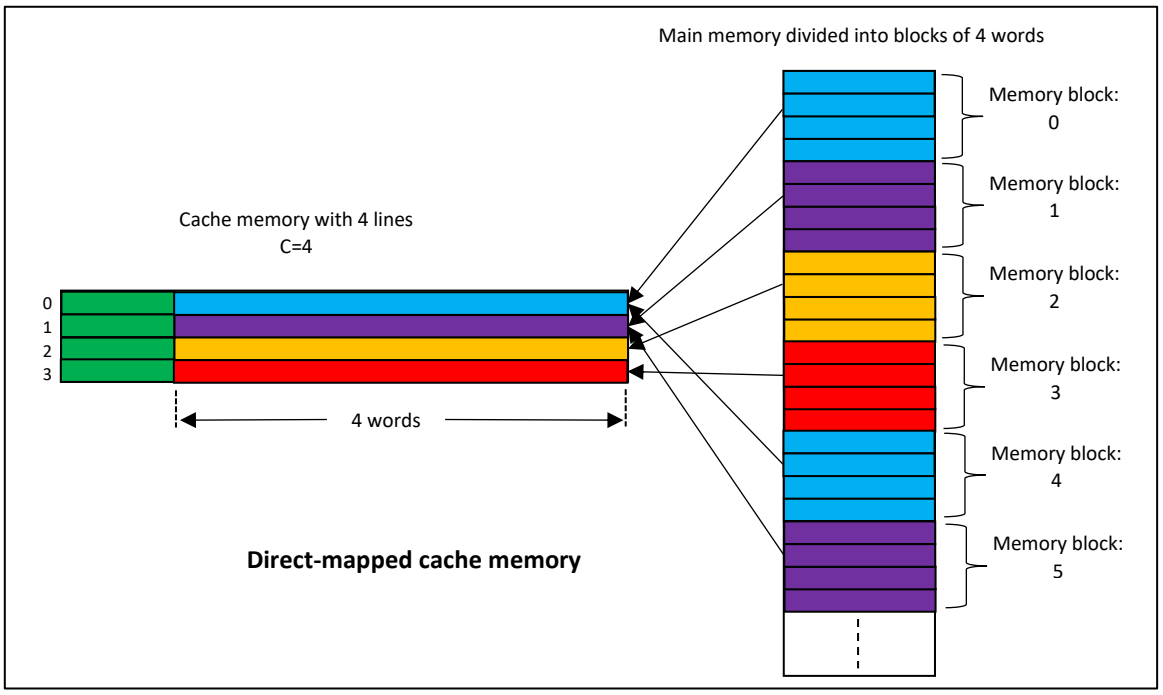


Figure 2.25: Illustrative image of direct-mapped cache memory

A ***memory address*** is divided into three fields (see Figure 2.26):

1. **Word** field, composed of **w** least significant bits, it determines the word searched (or byte) in the cache line among the $\mathbf{2^w}$ words.
2. **Line** field, determines the cache line where the memory block has been stored. If the cache memory consists of $\mathbf{2^r}$ cache lines, then this field is composed of **r** bits.
3. **Tag** field, identifies the memory block stored in the cache line. If the main memory contains $\mathbf{2^b}$ blocks, then this field will be composed of **b-r** bits.

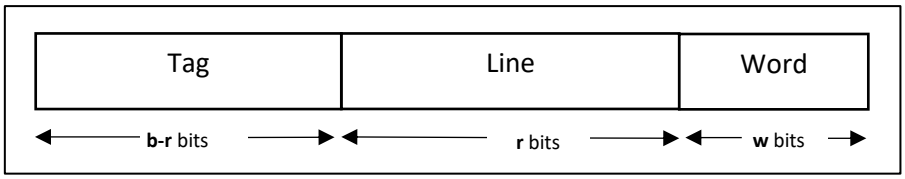


Figure 2.26: Main memory address format for direct mapping

Searching for a word in a direct cache (Figure 2.27) consists first of selecting the cache line using the bits of the **line** field, then if this entry is valid, its tag is compared with the one searched (the bits of the **tag** field), if there is a match between these two tags then there is a cache hit and the searched word is extracted from this line using the bits of the **word** field.

Otherwise, i.e., if the entry is not valid or the two labels do not match, a cache miss occurs.

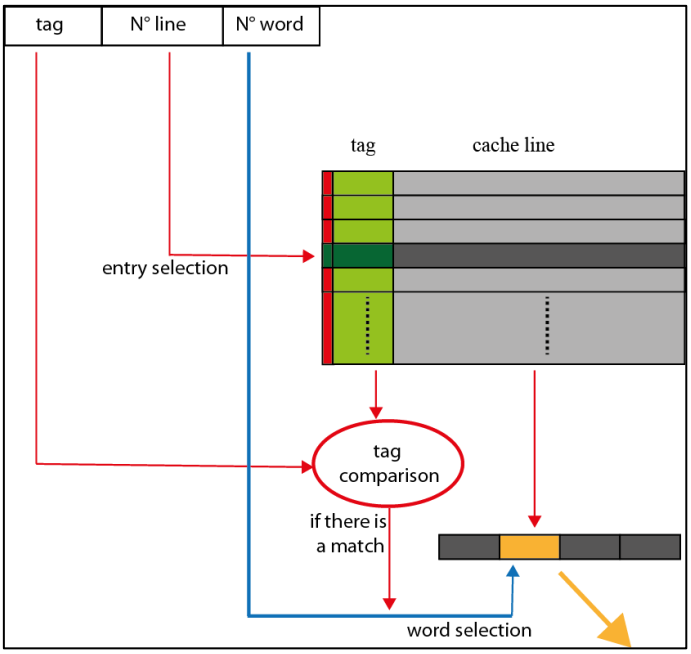


Figure 2.27: Searching for a word in a direct cache

Direct cache is simple and inexpensive to implement, but its main drawback is its limited flexibility; each memory block has only one location in the cache.

2.6.5.2 Associative mapping

In an associative cache, a memory block can be placed at any location in the cache memory (Figure 2.28).

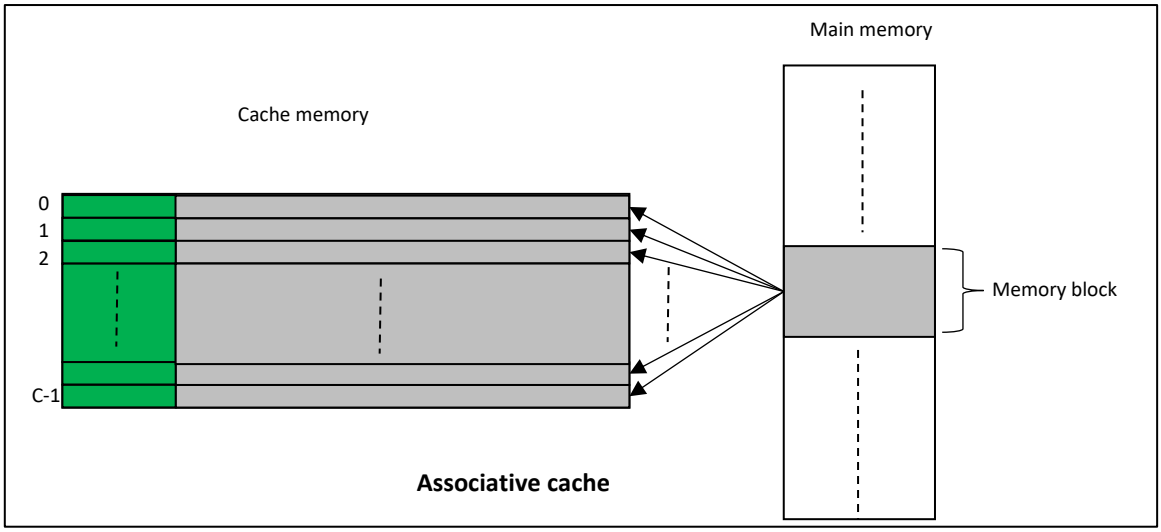


Figure 2.28: Illustrative image of the associative mapping

To search for a memory address in the cache, it is necessary to check all available locations. This consists of comparing, for each of them, the stored tag with that associated with the searched memory address.

The memory address is divided into two parts: the **w** least significant bits indicate the movement within the line, while the remaining **b** bits constitute the tag (see Figure 2.29).

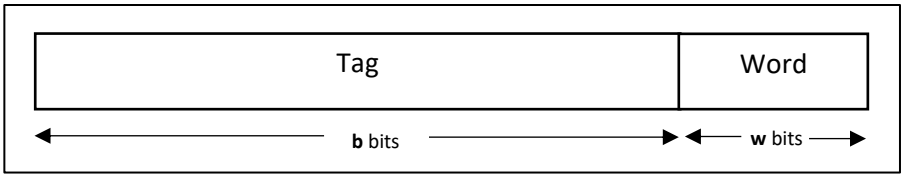


Figure 2.29: Main memory address format for associative mapping

The main drawback in fully associative cache memories is the complexity of the circuitry required to examine the tags of all cache lines in parallel.

2.6.5.3 Set-associative mapping

A good compromise between the two previous techniques can be found with set-associative mapping. Instead of restricting a memory block to a cache location (direct mapping) or allowing it to occupy all possible places (fully associative), we give ourselves limited freedom by allowing a memory line to be placed at any location among a precise set of possibilities (Figure 2.30).

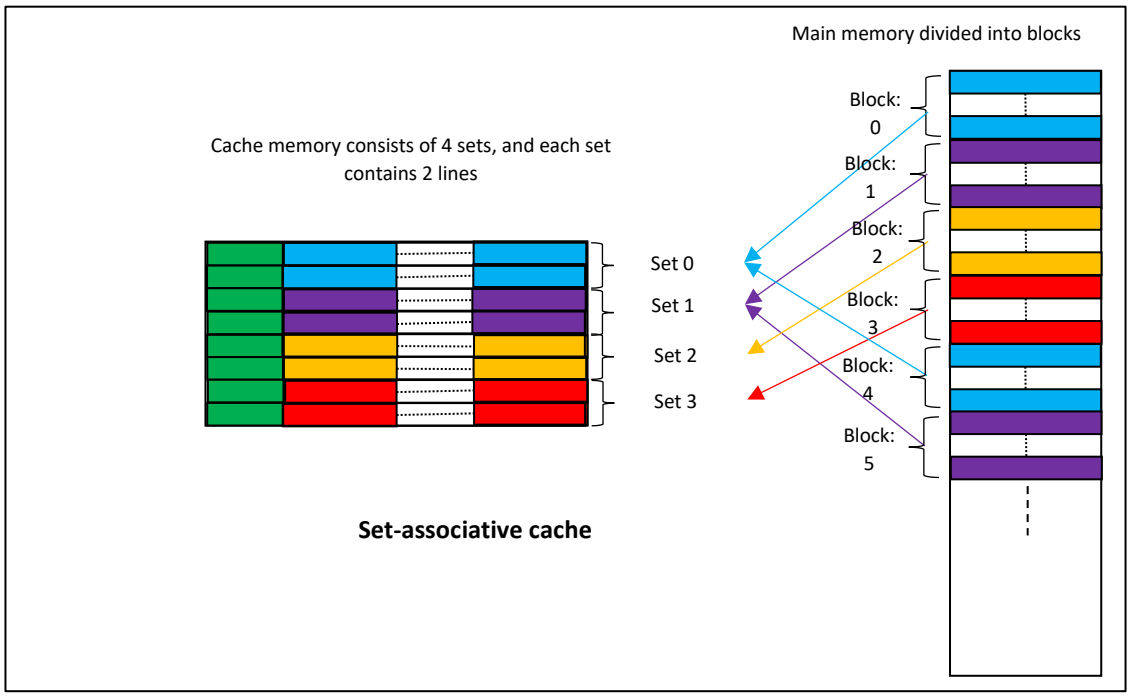


Figure 2.30: Illustrative image of set-associative mapping

If a cache memory consists of ***v*** sets and each set contains ***k*** possible locations for a memory block, then it is called a k-way set associative cache. The number of cache lines ***C*** constituting this memory is then:

$$C = v \times k$$

The number of the set ***i*** in which a memory block ***j*** will be stored is determined as follows:

$$i = j \ modulo \ v$$

i.e., block j will be stored in one location among the ***k*** possible locations of set ***i***.

Note that a 1-way set-associative cache is equivalent to a direct cache. A **C**-way set-associative cache of ***C*** lines in total is similar to a fully associative cache.

In this type of cache, the memory address is divided into three fields (see Figure 2.31):

1. **Word** field, composed of **w** least significant bits, it determines the word searched (or byte) in the cache line among the $\mathbf{2^w}$ words constituting this line.
2. **Set** field, determines the set where the memory block has been stored. If the cache memory is made up of $\mathbf{2^s}$ sets, then this field will be composed of **s** bits.
3. **Tag** field, identifies the memory block stored in a cache line of the selected set. If the main memory contains $\mathbf{2^b}$ blocks, then this field will be composed of **b-s** bits.

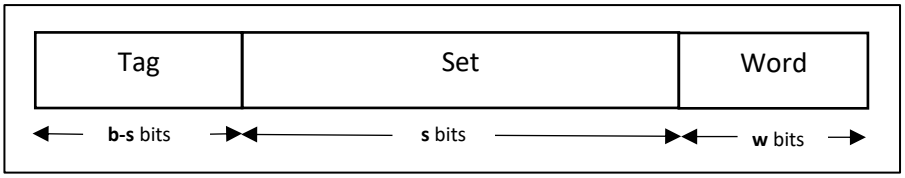


Figure 2.31: Main memory address format for set-associative mapping

2.6.6 Replacement algorithms

Once all the cache lines are occupied, adding a new memory block requires replacing one of the existing lines. In a direct-mapped memory, the line where the new block will be placed is strictly predetermined. On the other hand, in an associative or set-associative cache, multiple locations are possible for each block. Therefore, a replacement algorithm must be followed for selecting this victim line.

1. **First In First Out (FIFO)**: In this case, the replaced line is the one that has been loaded into memory for the longest time.
2. **Least Recently Used (LRU)**: this algorithm associates with each line in the cache information indicating the date of its last use. When space is needed to insert a new line, the algorithm selects and replaces the one whose last use goes back the furthest in time.
3. **Least Frequently Used (LFU)**: This algorithm associates with each line a counter that increments with each use. When a replacement is necessary, the victim line is the one with the lowest counter, hence the name. Although sorting counters is simpler than managing dates, this approach has a drawback: it can keep in the cache lines that were formerly popular (with a high counter), but have become obsolete, even if they are no longer needed by the program.

2.6.7 Writing/Reading Policy

Once the victim line is selected, it is necessary to decide what to do if this line has been modified and no longer matches its copy in main memory (dirty block).

There are two basic write policies:

2.6.7.1 Write through

We write simultaneously to the cache and main memory. We therefore guarantee consistency (Figure 2.32[5]).

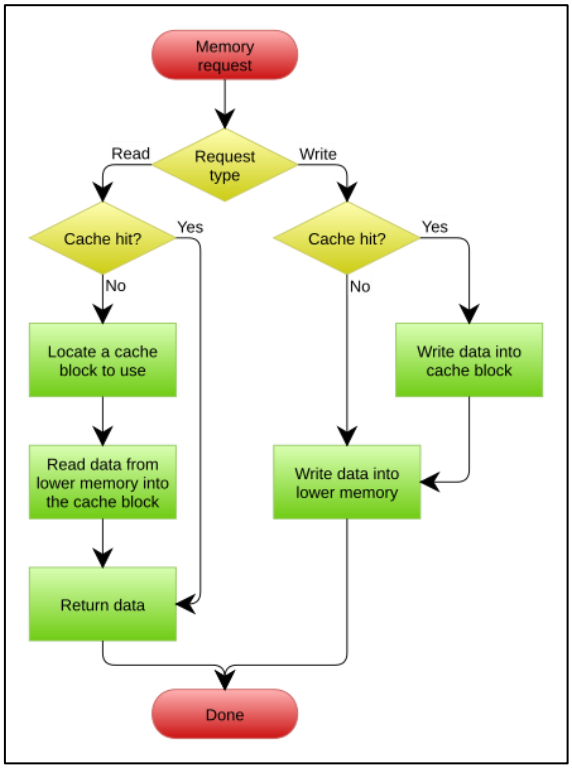


Figure 2.32: Write through (image from Wikipedia)

2.6.7.2 Write back

There are several techniques here again. The central memory can be updated when the information in the cache memory needs to be replaced or as soon as the communication bus is free. In these techniques, consistency is not guaranteed permanently, but the writing time is lower (Figure 2.33[6]).

[5] https://fr.wikipedia.org/wiki/M%C3%A9moire_cache#/media/Fichier:Write-through_with_no-write-allocation.svg
[6] https://fr.wikipedia.org/wiki/M%C3%A9moire_cache#/media/Fichier:Write-back_with_write-allocation.svg
(last accessed: 18/01/2025 at 18 :40)

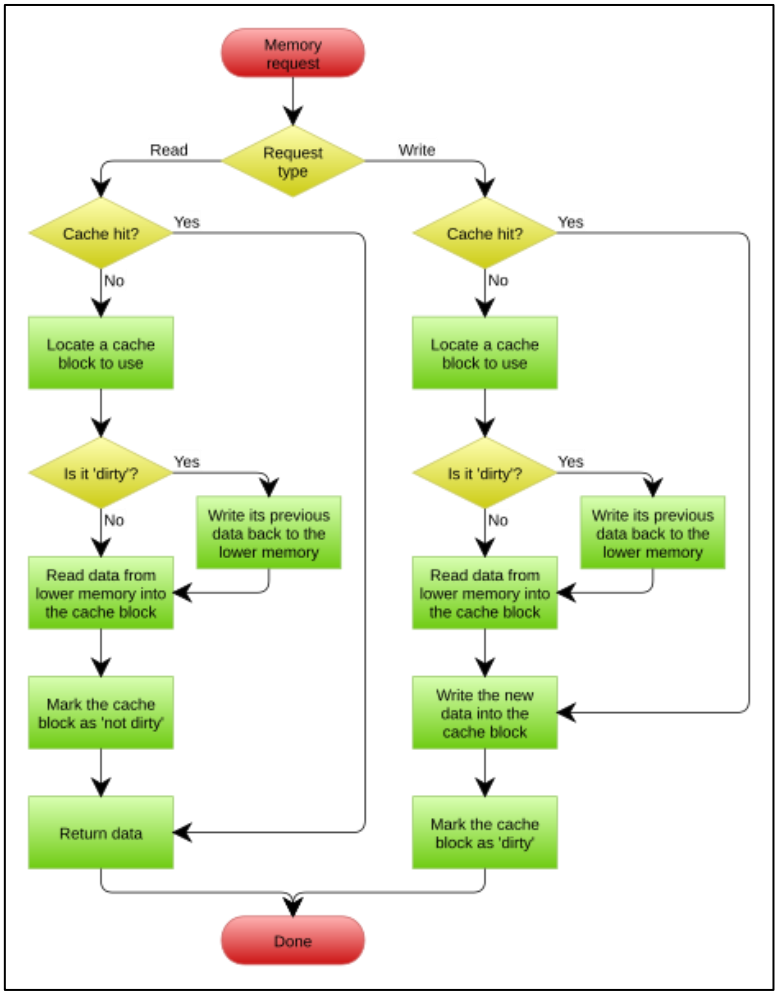


Figure 2.33: Write back (image from Wikipedia)

Chapter 3. Notions on computer instructions

3.1 HIGH-LEVEL LANGUAGE, ASSEMBLER, MACHINE LANGUAGE 45
3.1.1 Machine language 45
3.1.2 Assembly language 45
3.1.3 High-level language 46
3.2 NOTIONS ON INSTRUCTION SETS 47
3.3 ADDRESSING MODES 48
3.4 THE DIFFERENT STAGES OF GENERATING AN EXECUTABLE PROGRAM 49
3.5 COMPILATION 49
3.6 ASSEMBLING PROCESS 50
3.7 CONTROL UNIT 51
3.8 STEPS FOR EXECUTING AN INSTRUCTION 52
3.9 UCC PIPELINE 53
3.9.1 Pipelining technique 53
3.9.2 Pipeline performance 54
3.9.2.1 Speed-up: 54
3.9.2.2 Throughput: 54
3.9.3 Some examples of pipeline depth 55
3.9.4 Pipeline Hazards 55
3.10 CLOCK 56
3.11 SEQUENCER 56
3.11.1 Hardwired control 57
3.11.1.1 Advantages: 57
3.11.1.2 Disadvantages: 57
3.11.2 Microprogrammed control 57
3.11.2.1 Advantages: 57
3.11.2.2 Disadvantages: 58

3.1 High-level language, assembler, machine language

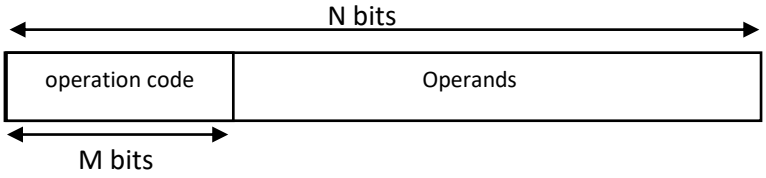
Programming consists of transforming an algorithm into a program that can be understood by a computer, using an appropriate language. This activity can be carried out at different levels of abstraction, more or less close and dependent on the physical architecture of the machine. Essentially, we will distinguish three levels:

- ✓ Low-level programming in machine language;
- ✓ Low-level programming in assembly language;
- ✓ High-level programming using a high-level language or advanced language.

3.1.1 Machine language

At this level, programming and instructions are closely tied to the machine architecture and the processor. They operate directly on the processor registers and manipulate physical memory addresses, all in the form of binary strings. A machine language instruction is usually presented as a binary string divided into two main parts:

- ➢ The ***operation code*** specifies the type of operation to be performed, such as an addition, a logical operation or a memory read, etc.
- ➢ ***Operands*** represent the data on which the operation defined by the operation code will be performed. These operands can be words located in memory, processor registers or immediate values integrated directly into the instruction.



3.1.2 Assembly language

Assembly language is a symbolic version of machine language, allowing the programmer to manipulate machine instructions, in particular without binary codes and address calculations. While remaining specific to the machine architecture, assembly language retains the same set of instructions as machine language.

An assembly language instruction is structured in several fields, separated by one or more spaces:

- ➢ A ***label*** field (optional), it represents the address of the machine instruction.
- ➢ An ***operation code*** field, it corresponds to the binary code of the operation in machine language.
- ➢ An ***operand*** field, it contains one or more operands, separated by commas. These operands represent registers, memory words or immediate values, appearing in machine instructions.

Example of MIPS assembly language instruction:

```
Label operation code operands
Boucle : add $s0, $s1, $s2
```

Although programming in assembly language is easier than at the machine level, it remains laborious for a human being. It mainly requires knowledge of the architecture of the processor and the machine. For example, we speak of the assembly language of the Intel Pentium processor or the assembly language of the AMD Athlon processor.

To execute a program written in assembly language, it is essential to translate its instructions into equivalent machine language. This conversion is carried out by a specific tool called ***assembler***.

3.1.3 High-level language

These languages are distinguished by their total independence from the architecture of the machine and the processor. In addition, they offer a richer expressiveness and are better adapted to human logic, thus making it easier to translate the algorithms established to solve a problem.

Examples of high-level languages: C, C++, C#, Delphi, Java, etc.

High-level languages are divided into several large families. Mainly, two families of languages are distinguished: ***procedural languages*** and ***object languages***.

- **Procedural language**: Writing a program is based on the notions of procedures and functions, which define the processing to be applied to the data to solve the initial problem. The *C* and *Pascal* languages are two examples of procedural languages.
- **Object language**: writing a program is based on the notion of objects, representing the entities involved in solving the problem. Each of these objects has methods attached to them, which when activated, modify the state of the objects. *Java* and *Eiffel* are two examples of object-oriented languages.

3.2 Notions on instruction sets

The instruction set of a machine is the set of instructions that can be executed by the machine. The instructions are of different types:

A. **Arithmetic and logic instructions**:
   These instructions represent the operations that can be performed by the Arithmetic and Logic Unit (ALU). Among the instructions of this type: ADD, SUB, AND, XOR…
   The PSW (Processor Status Word) register is updated after the execution of an instruction of this type.

B. **Data transfer instructions**:
   These instructions are used to move data between registers, between registers and memory cells and between memory cells. Examples of this type of instructions: LOAD, STORE, MOV.

C. **Branch instructions**:
   These instructions allow jumps to be made in the program code, to a given instruction. Branches can be *conditional* or *unconditional*.
   1. **Conditional branch instructions** jump to a given address if a condition is met. These conditions are related to the flags in the PSW status register.
      Example: JZ, JNE…
   2. **Unconditional branch instructions** jump to a given address unconditionally. That is, the jump is always performed.
      Example: JMP…

D. **Subroutine call instructions**:
   A subroutine is a group of instructions that perform a specific task. When a program needs a task, it calls the corresponding subroutine (CALL). At the end of the subroutine execution, the subroutine must return to the main program (RET).
   - Function call
     - Passing parameters
       - ✓ By register: limited number (e.g.: 4 for MIPS)
       - ✓ By stack: additional parameters
     - Saving the return address
       - ✓ In a general register: R31 (MIPS)
       - ✓ In a specific register: LR (PowerPC)
       - ✓ In the stack: IA-32
   - Return
     - Retrieving the result
       - ✓ By registers
       - ✓ By stack
     - Instruction loading the return address into PC
       - ✓ For example: JMP R31 (MIPS)

3.3 Addressing modes

Addressing modes are used to determine the location of the operands of an instruction. The processor identifies the mode used in a given instruction using the opcode or through a field in the instruction format, reserved for this purpose. In practice, there are several addressing modes, among others, we cite:

1. **Immediate addressing**: in this addressing mode, the operand is immediately available in the instruction itself (Figure 3.1-(1)).
2. **Direct addressing**: the effective address of the operand is directly specified in the address field of the instruction (Figure 3.1-(2)).
3. **Indirect addressing**: in this mode, the address field of the instruction contains the address of a memory word, and the latter contains the effective address of the operand (Figure 3.1-(3)).
4. **Register addressing**: in this mode, the operand is located in the register whose identifier is specified in the instruction (Figure 3.1-(4)).
5. **Register indirect addressing**: in this mode, the effective address where the operand is located is indicated in a register (Figure 3.1-(5)).
6. **Displacement Addressing**: the effective address of the operand is the sum of the address contained in a register (register indirect) and an immediate displacement indicated in the instruction itself (Figure 3.1-(6)).

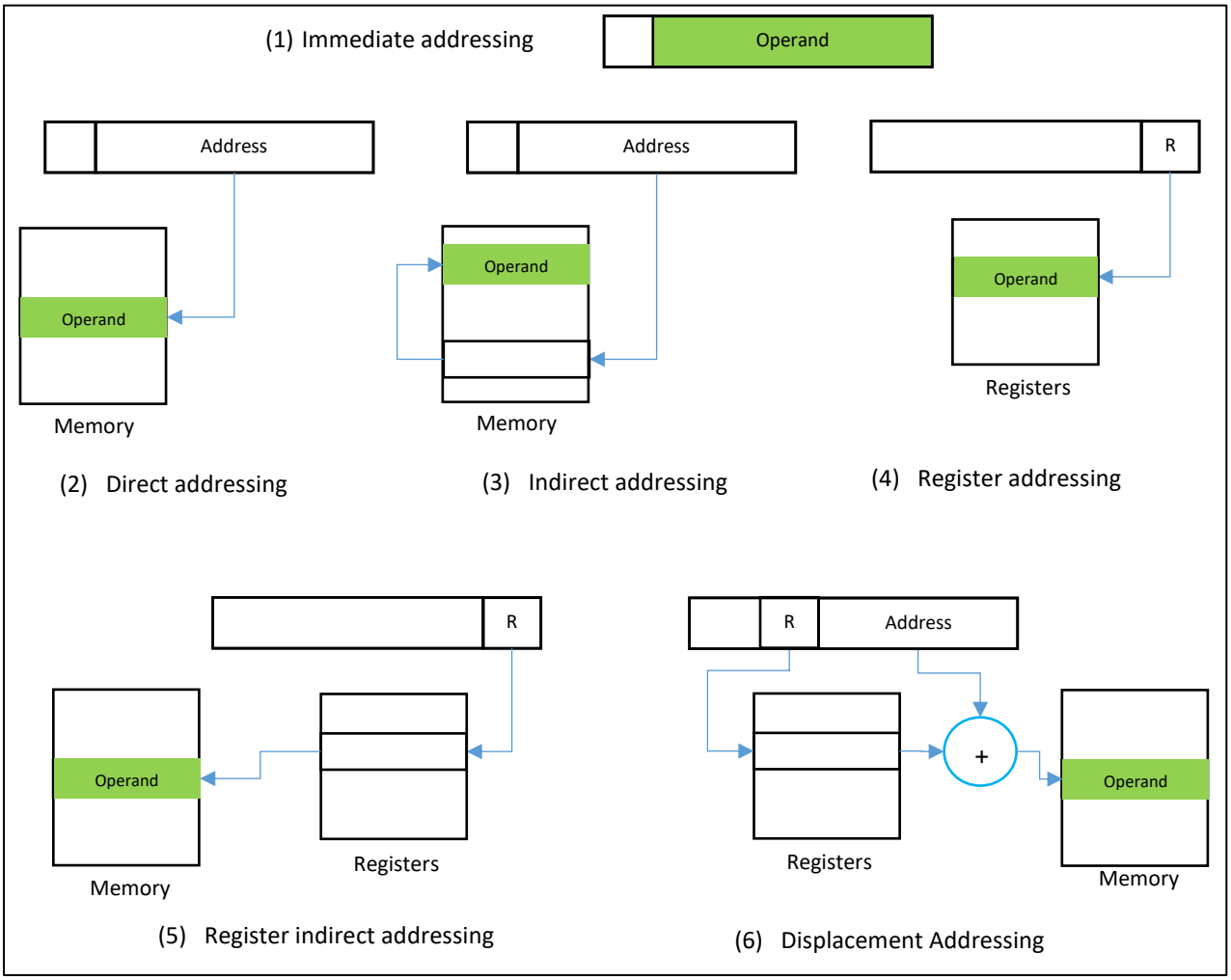


Figure 3.1: Illustration of some addressing modes

3.4 The different stages of generating an executable program

Generating an executable program from source code involves a set of steps (Figure 3.2). In each step a particular program is used:

1. **Compiler**: is a program that translates a source code written in high-level language into machine language or assembly language.
2. **Assembler**: is a program that translates a source code written in assembly language into machine language.
3. **Linker (**also called **link Editor)**: is a program that combines object files into an executable file, usually stored on disk.
4. **Loader**: is a program that belongs to the operating system. Its role is to load the executable program into memory.

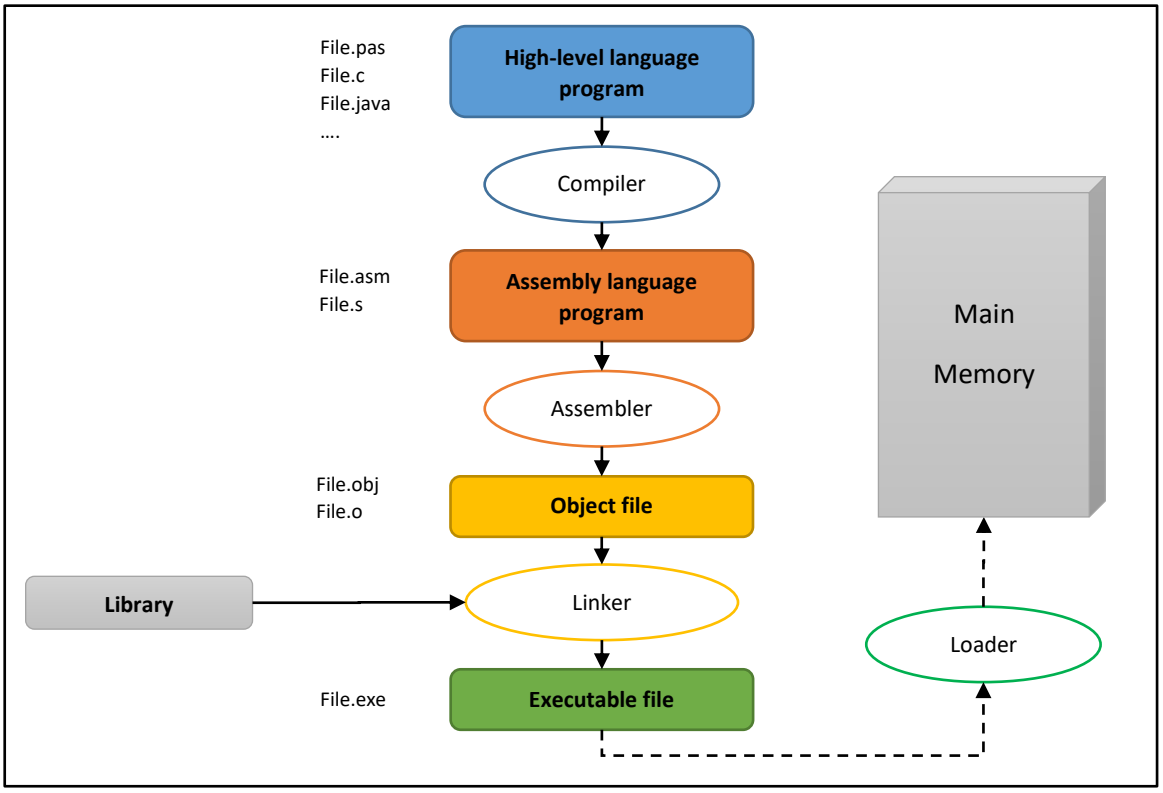


Figure 3.2: Program production line

3.5 Compilation

Compilation is the process of translating a source program written in a high-level language into a program called an ***object program***, stored on disk. This operation is performed by a specific tool called a ***compiler***. The compiler depends on the physical machine to which it must produce the language.

Compilation takes place in several phases:

1. **Lexical analysis**: The *lexical analyzer* (or *scanner*) reads the source program character by character to identify the ***lexemes***[7] of the language. It eliminates insignificant spaces and comments, which have no role in the logical structure of the program. The recognized lexemes are then translated into intermediate symbols that are easier to handle (for example, integers).

[7] A lexeme is a group of elementary symbols.

2. **Syntactic analysis**: The *syntactic analyzer* (or *parser*) checks whether the sequence of lexical units produced by the lexical analysis complies with the syntactic rules of the language.
3. **Semantic analysis**: The semantic analysis performs a number of semantic checks, including:
    - **Declaration verification**: ensuring that all used variables have been previously declared.
    - **Type checking**: verifying that the types of the operands of an operator are compatible with the operation performed.
    - **Automatic type conversions**.

3.6 Assembling process

The operation of translating an assembly language program into a machine language program (Figure 3.3) is performed by the ***assembler***. This process generally takes place in two passes:

1. First pass: the assembler goes through each line of the file and breaks it down into lexemes. If the line begins with a label, it records in a symbol table the name of the label as well as the memory address associated with the corresponding instruction.
2. Second pass: the assembler uses the symbol table established during the first phase to generate the machine code. In fact, it reads each line of the file again and if the line contains an instruction, it combines the binary representation of its opcode and its operands.

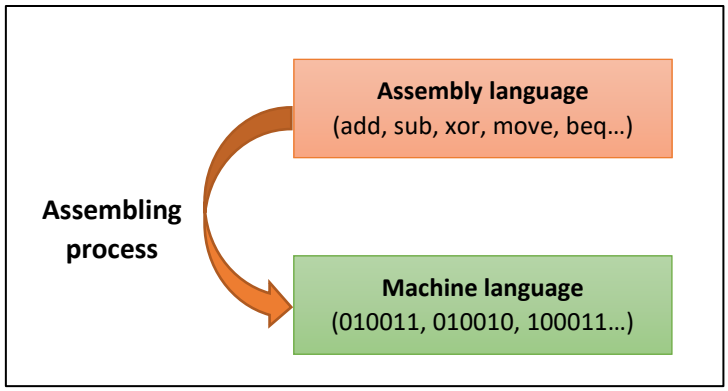


Figure 3.3: Assembling process

3.7 Control unit

Control Unit (CU) is responsible for executing machine instructions, it controls the movement of data and instructions, as well as the operations of the arithmetic and logic unit. It represents the **control part**.

This unit contains a decoder, a sequencer and two registers for handling instructions, the Instruction Register (IR) and the Program Counter (PC) (Figure 3.4):

a. **Decoder:** is a circuit whose function is to identify the instruction found in the IR. The decoder takes as input the bits that constitute the operation field and activates a single line (output) that corresponds to the decoded instruction. Each instruction in the processor instruction set is associated with a line.
b. **Sequencer:** is a set of circuits whose function is the effective execution of the instruction found in the IR. The sequencer executes, timed by the microprocessor clock, a sequence of microcommands (microinstructions) performing the work associated with this machine instruction (see Sequencer in the section 3.11).
c. **Program Counter (PC)**: is a register that contains the address of the next instruction to be executed.
d. **Instruction Register (IR)**: is a register that contains the instruction to be executed.

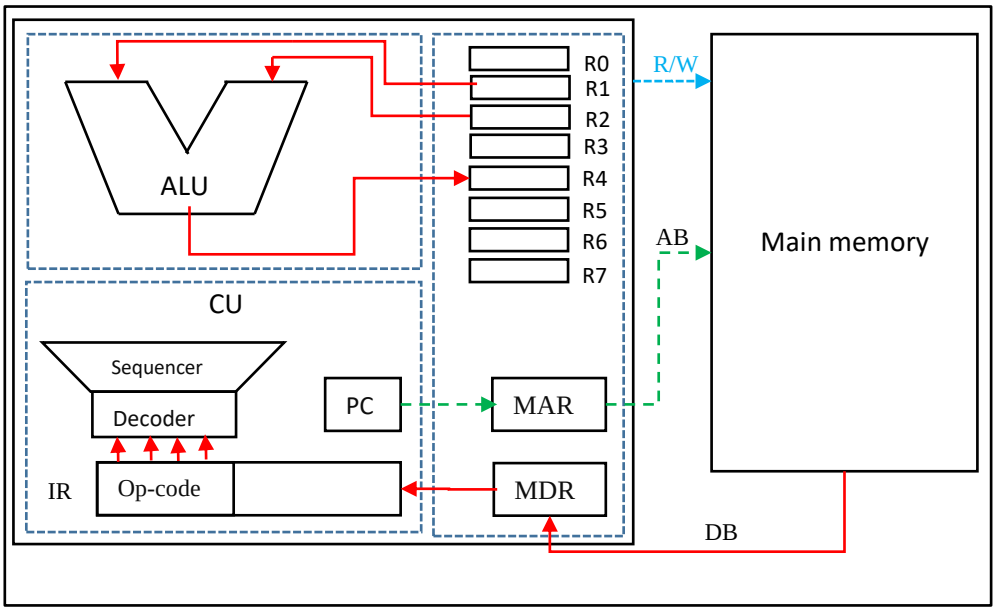


Figure 3.4: Central processing unit main components with the memory

- **Memory Data Register (MDR)**: It allows the exchange of information (content of a memory word) between the main memory and the processor (register).
- **Memory Address Register (MAR)**: It is connected to the address bus (AB) and allows the selection of a memory word via the selection circuit.

3.8 Steps for executing an instruction

A **program**: is a sequence of elementary instructions ordered and executed by the processor.

The different stages of executing an instruction are as follows:

1. Find the next instruction to execute from memory and load it into the instruction register.
   MAR ← PC;
   MDR ← Memory [MAR];
   IR ← MDR;
2. Increment the value of the program counter so that it points to the next instruction.
   PC ← PC + 1;
3. Determine the type of instruction loaded.
   Decoding (IR);
4. If the instruction uses data that is in memory, determine where it is and load it into one of the CPU registers.
5. Execute the instruction.
6. Go to step 1 to begin executing a new instruction.

This instruction sequence is known by the **Fetch/Decode/Execute** cycle.

3.9 UCC Pipeline

The need for high-performance computers has always been a necessity and remains so. Therefore, computer designers are constantly trying to optimize the performance of machines. To this end, they use several techniques, including increasing the clock frequency and using multiple processors operating in parallel. In this section, we present a technique of parallelism at the instruction level, namely ***pipelining***.

3.9.1 Pipelining technique

Pipelining refers to a technique where a task is divided into multiple subtasks that are executed sequentially. Each is executed by a specific functional unit. One task is accepted as input before the previous task is output from the other end, which improves performance over sequential processing.

Figure 3.5 illustrates the difference between sequential and pipeline processing of three instructions. In this example, an instruction is divided into four subtasks (F, D, E, and W). In other words, the execution of an instruction in this example goes through four stages. The sequential processing of these three instructions required 12-time units. However, only 6-time units were required for the execution of these three pipelined instructions.

The total execution time ***T*** of ***n*** instructions in a ***k***-stage pipeline is given by the following formula:

$$T_{k,n} = [k + (n-1)]\tau$$

Note that the sequential execution time of these ***n*** instructions is $(n \times k)\tau$. For example, in Figure 3.5 below, we have 3 instructions (n=3), each of which is executed in 4-time units (τ) (i.e., k=4). Which makes the execution time T: $T = (n \times k)\tau = (3 \times 4)\tau = 12\tau$. However, the execution of these 3 instructions in a 4-stage pipeline (k=4) completes in 6-time units:

$$T_{k,n} = [k + (n-1)]\tau = [4 + (3-1)]\tau = 6\tau$$

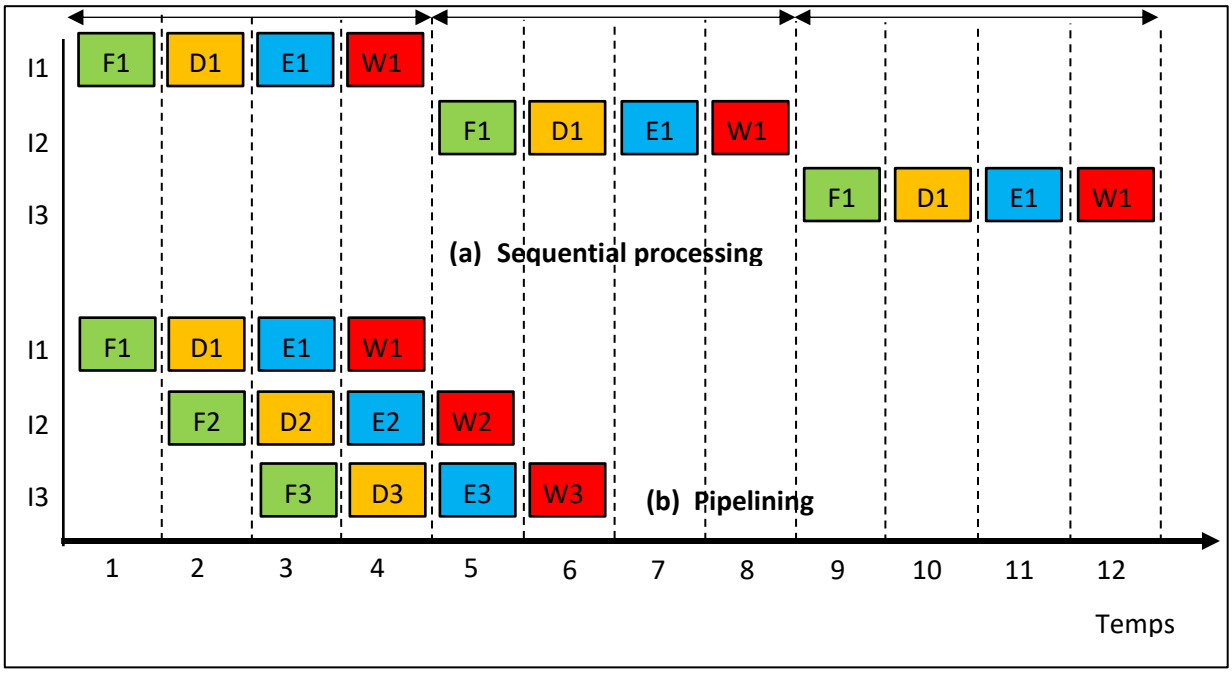


Figure 3.5: Difference between sequential processing (a), and pipeline processing (b)

It is possible to represent the pipeline in another way. This time, the vertical axis represents the subtasks and the horizontal axis the time. Figure 3.6 below illustrates this second representation.

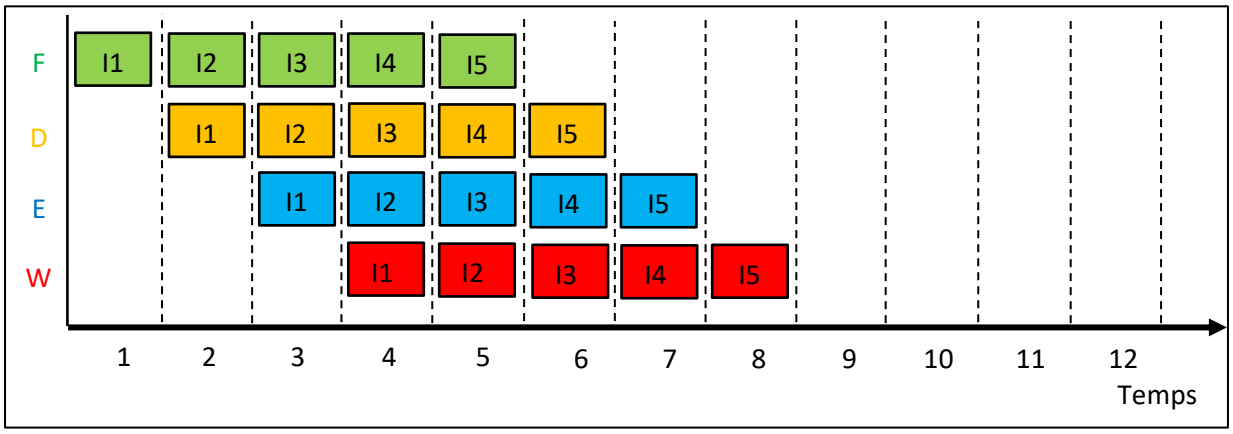


Figure 3.6: Second pipeline representation; the vertical axis represents the subtasks.

Note that in the example in the previous figure (Figure 3.6), there are five instructions: I1, I2, I3, I4, and I5 (i.e., n=5) and still four stages (k=4). The execution of these five instructions completes in 8-time units.

$$T_{k,n} = [k + (n-1)]\tau = [4 + (5-1)]\tau = 8\tau$$

In fact, the pipeline does not improve the execution time of an instruction, but rather it improves the instruction throughput. That is, the number of instructions executed during the same duration of time.

3.9.2 Pipeline performance

In this section, we present two performance measures to evaluate the quality of a pipeline.

3.9.2.1 Speed-up:

The Speed-up S_k that we obtain by using a ***K***-stage pipeline is calculated by the following formula:

$$S_k = \frac{Time\ using\ sequential\ processing}{Time\ using\ pipeline\ processing} = \frac{(n \cdot k)\tau}{[k + (n-1)]\tau} = \frac{n \cdot k}{k + (n\ -\ 1)}$$

When the number of instructions *n* tends to infinity, the limit of S_k is *k*:

$$\lim_{n \to \infty} S_k = k$$

i.e., the increase in speed is theoretically k times.

3.9.2.2 Throughput:

Throughput D_k, the number of instructions executed per unit time using a *K*-stage pipeline is calculated by the following formula:

$$D_k = \#\ instructions\ executed\ per\ unit\ of\ time = \frac{n}{[k + (n-1)]\tau}$$

When the number of instructions *n* tends to infinity, the limit of D_k is *1* (assuming that $\tau = 1$):

$$\lim_{n \to \infty} D_k = 1$$

3.9.3 Some examples of pipeline depth

The number of stages in a pipeline varies from computer to computer. For example, the MIPS R2000 and R3000 processors have 5-stage pipelines. Other processors may have 31-stage pipelines, such as the *Intel Pentium 4 Prescott*.

The table below shows the pipeline depth of some processors[8].

Table 3.1: Pipeline depths of some processors

Processor	Depth
Intel Pentium 4 Prescott	31
Intel Pentium 4	20
IBM PowerPC 970	16
IBM POWER5	16
AMD K10	16
Sun UltraSPARC IV	14
Sun UltraSPARC IIi	14
Intel Pentium II	14
Intel Core 2 Duo	14
IBM POWER4	12
AMD Opteron 1xx	12
AMD Athlon	12
Intel Pentium III	10
Intel Itanium	10
MIPS R4400	8
Motorola PowerPC G4	7

3.9.4 Pipeline Hazards

Ideal operation of the pipeline is not guaranteed, as it may be subject to disruptions due to several events called ***pipeline hazards***.

Situations where the next instruction cannot execute on the next clock cycle fall into three main types:

1. ***Structural hazards (resource hazards)***: This type of hazard is caused by conflicts in accessing a hardware resource such as register, memory, ALU, etc. In other words, the hardware cannot handle the combination of instructions that we want to execute in the same clock cycle. For example, trying to access a register or memory by two instructions in the same clock cycle.
2. ***Data hazards (data dependency)***: This type of hazard is caused when the data required for the execution of a scheduled instruction is not yet available and therefore it cannot be executed in the scheduled clock cycle.
3. ***Control hazards*** (also called ***branch hazards***): This type of hazard is caused by the presence of branch instructions. The instructions that follow have already been loaded into the pipeline at the time they are to be executed. However, if this is a "simple" branch, the following instruction should absolutely not be executed.

[8] https://fr.wikipedia.org/wiki/Pipeline_(architecture_des_processeurs) (last accessed: 25/10/2024 at 15 :30)

3.10 Clock

Executing an instruction involves several distinct steps, including retrieving the instruction from memory, decoding its various components, loading and storing data, and performing logical and arithmetic operations. These steps are timed by a system *clock*.

A clock is a circuit that emits a series of pulses with a precise pulse width and a precise interval between consecutive pulses. The clock is therefore characterized by its *frequency*[9], measured in MHz, where 1 MHz corresponds to 1 million cycles per second and 1 GHz corresponds to 1 billion cycles per second, so 1 Hz corresponds to one cycle per second.

The processor requires a specific number of clock cycles to execute each instruction. Thus, instruction performance is usually measured in clock cycles rather than seconds (see Performance in Chapter 4). A clock cycle (or clock period) is the time between two pulses (Figure 3.7), and it is the reciprocal of the clock frequency.

For example, 2.5 ns is the clock cycle time of a processor clocked at 400 MHz.

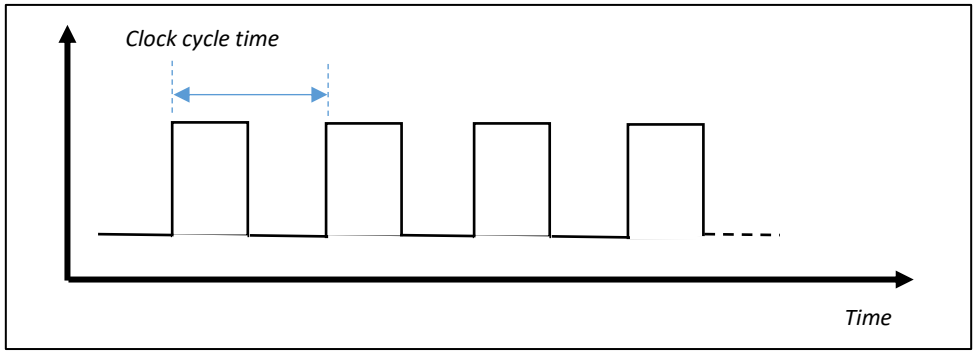


Figure 3.7: Illustrative image of a clock cycle

3.11 Sequencer

The sequencer is an automaton responsible for generating microcommands (Figure 3.8). This sequential machine operates based on several inputs, including:

- ✓ The instruction code (or the outputs of the decoder if a distinction is made between the decoder and the sequencer)
- ✓ Information about the processor's state, particularly flag status.

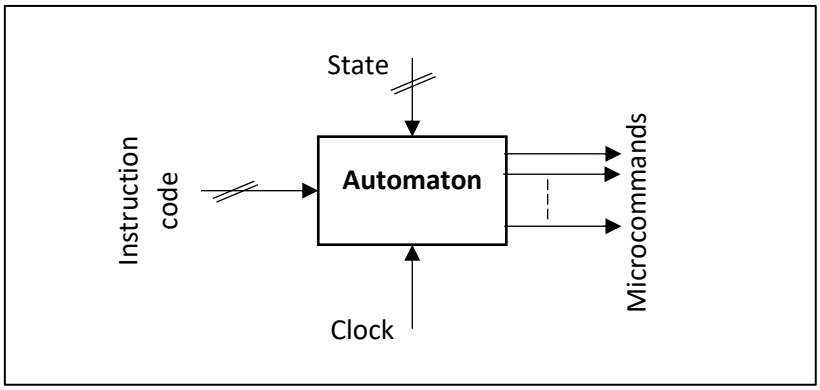


Figure 3.8: The sequencer

[9] The clock frequency, also called ***clock rate*** or ***clock speed***.

- ✓ The state of the control bus lines.
- ✓ Clock signals.

There are two types of control units, ***hardwired*** and ***micro-programmed***.

3.11.1 Hardwired control

A hardwired sequencer is composed of combinational circuits that generate the set of commands that correspond to the instruction being executed as determined by the operation code.

3.11.1.1 Advantages:

The advantage of the wired sequencer is that it is very fast.

3.11.1.2 Disadvantages:

The disadvantage is that the instruction set and control logic are directly linked by specialized circuits that are complex and difficult to design or modify.

3.11.2 Microprogrammed control

The microprogrammed sequencer is structured around several key elements: a memory called microprogram memory (also called ***control memory***), an arithmetic unit, a microcounter, a circuit responsible for generating the initial address of the instruction, as well as a circuit responsible for triggering the execution of a micro-instruction.

The general structure of the microprogrammed sequencer is depicted in Figure 3.9.

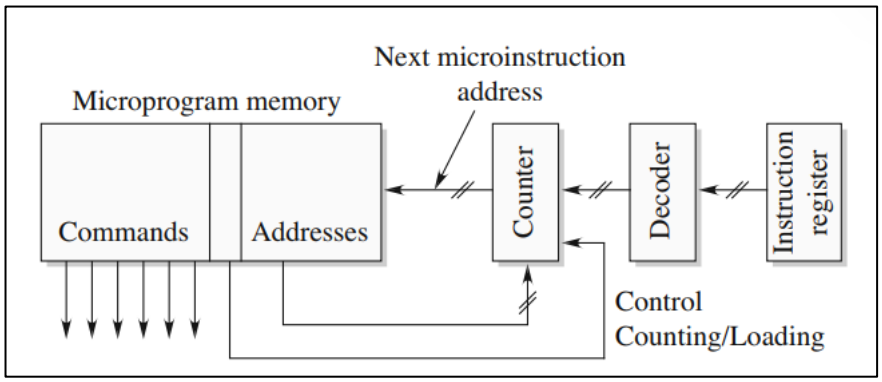


Figure 3.9: General structure of the microprogrammed sequencer [6]

Each bit of a ***microinstruction*** corresponds to a ***control signal***: a bit at **1** indicates that the signal is active (and thus the corresponding ***microoperation*** is to be initiated), while a bit at **0** means that the signal is inactive (and thus the corresponding microoperation remains idle). In other words, each microinstruction is a string of 0's and 1's called a ***control word***, thus each word in control memory contains within it a microinstruction. Each binary element of a word in memory indicates the state of one of the microoperations (also called ***microcommands***) in the *phase* (Figure 3.10). A sequence of microinstructions constitutes a ***microprogram*** (or ***firmware***). Generally, the control memory is a read-only memory (ROM).

3.11.2.1 Advantages:

Microprogramming is flexible, simple in design, and allows for the creation of very powerful instruction sets.

3.11.2.2 Disadvantages:

The disadvantage of this approach is that all instructions must go through an additional level of interpretation, which slows down program execution.

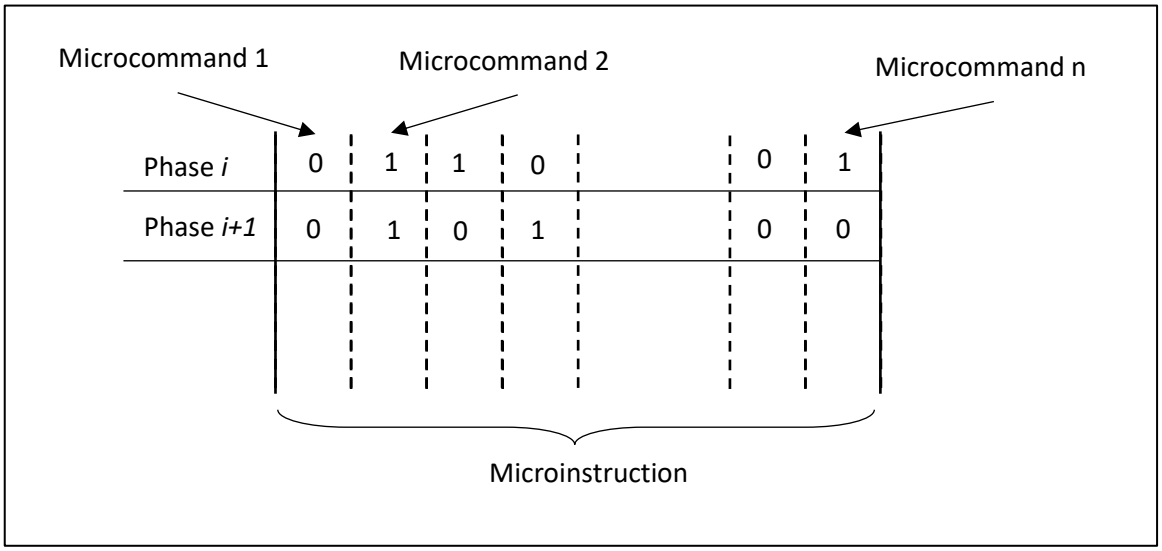


Figure 3.10: Microcommands

Chapter 4. Processor

4.1 PROCESSOR ROLE 60
4.2 PERFORMANCE 60
4.3 MEASURING PERFORMANCE 60
4.4 CYCLE PER INSTRUCTION (CPI) 60
4.5 CPU EXECUTION TIME 61
4.6 NOTIONS ON CISC AND RISC ARCHITECTURES 62
4.6.1 Complex instruction Set Computer (CISC) *62*
4.6.2 Reduced instruction Set Computer (RISC) *62*
4.7 PRESENTATION OF THE MIPS R3000 MICROPROCESSOR 63
4.8 EXTERNAL STRUCTURE OF THE MIPS R3000 PROCESSOR 63
4.8.1 Registers *63*
4.8.1.1 CPU Registers 63
4.8.1.2 System control co-processor (CP0) registers 64
4.8.2 The Arithmetic and Logic Unit (ALU) *64*
4.8.3 The Control Unit (CU) *64*
4.8.4 Memory *64*
4.8.4.1 General memory layout 65
4.8.5 Instruction Register (IR) *66*
4.8.5.1 R-Format 66
4.8.5.2 I-Format 66
4.8.5.3 J-Format 67
4.9 INTERNAL STRUCTURE OF THE MIPS R3000 PROCESSOR 68
4.9.1 Processor Interface *68*
4.9.2 Internal architecture of the processor *68*
4.10 MIPS R3000 INSTRUCTION SET 69
4.10.1 Encoding MIPS instructions *69*
4.10.2 MIPS Instruction Set *71*
4.10.2.1 Arithmetic and Logic Instructions 71
4.10.2.2 Data transfer instructions (read/write Memory) 73
4.10.2.3 Branch instructions 74
4.11 MIPS PROCESSOR ADDRESSING MODES 75
4.12 PROGRAMMING IN MIPS ASSEMBLY LANGUAGE 77
4.12.1 Translation of some control structures *77*
4.12.1.1 If ... then …else 77
4.12.1.2 While 77
4.12.1.3 For 78
4.12.2 Assembler Directives *78*
4.12.2.1 Structure of a MIPS assembly language program 79
4.12.2.2 Data declaration 79

4.1 Processor role

The processor (CPU) fetches instructions from memory, reads and writes data to it, and transfers data to and from input/output devices. As we saw in the section 3.8, the execution cycle of an instruction takes place in several stages (fetch, decoding, etc.). This execution cycle repeats itself as long as there are still instructions to execute. During an execution cycle, the actions of the CPU are determined by micro commands generated by the control unit. These micro commands correspond to individual control signals transmitted by dedicated control lines. The activation of the control signals is determined either by hardwired control or by microprogramming (see section 3.11).

4.2 Performance

There are several factors that influence the processor execution time, and there are also several performance measures. The interest of performance measures depends on the quality of the user (simple or designer), each is much more interested in a particular type of measure. For example, simple (classic) users are generally interested in the response time of their machines. On the other hand, machine designers may rather be interested in the throughput or the number of cycles per instruction for example.

In fact, performance analysis should help us answer questions such as: *how fast can a program p run on a machine x*? Or, *how much more efficient is a machine A than another B*?...

- ✓ **Response time** (or **execution time**): this is the time between the start and the end of a task.
- ✓ **Throughput** (or **bandwidth**): this is the number of tasks completed per unit of time.

Improving (maximizing) performance means reducing response time:

$$Performance_X = \frac{1}{Execution\ time_x}$$

A computer **x** is said to be **n** times faster than a computer **y** if:

$$\frac{Performance_x}{Performance_y} = n$$

Or in other words, the execution time on computer **y** is **n** times longer than that on computer **x**:

$$\frac{Execution\ Time_y}{Execution\ Time_x} = n$$

4.3 Measuring performance

- ✓ **CPU execution time** (or **CPU time**): “It is the time that the CPU spends calculating for a specific task and does not include the time spent waiting for I/O or executing other programs”.
- ✓ **Clock cycle**: "The time for one clock period, usually the processor clock, that is running at a constant rate."
  - **Clock period**: The length of each clock cycle. i.e., the time of one complete clock cycle.
  - **The clock rate**: this is the inverse of the clock period.

$$Clock\ cycle\ time = \frac{1}{Clock\ rate}$$

4.4 Cycle per instruction (CPI)

Cycle per instruction (CPI): This is the average number of clock cycles per instruction for a program or program fragment.

$$CPU\ clock\ cycles = \#\ instructions\ for\ a\ program \times CPI$$

If a program is composed of ***n*** classes of instructions with C_i is the number of instructions of class ***i*** in this program and CPI_i is the ***CPI*** of the instructions of class ***i***. The total number of cycles is therefore:

$$Total\ number\ of\ clock\ cycles = \sum_{i}^{n}(CPI_i \times C_i)$$

So, the average number of cycles per instruction is:

$$CPI\ = \frac{\sum_i^n (CPI_i \times C_i)}{\sum_i^n C_i}$$

Example:

Consider a machine A for which the following performance measurements were recorded while running a set of benchmark programs. Assume that the processor clock frequency is 200 MHz.

Calculate the overall CPI of this machine.

Instruction classes	Percentage of occurrences	CPI
ALU	38	1
Load & store	15	3
Branch	42	4
Others	5	5

In this example, we have 4 categories of instructions (n=4). Therefore, the overall CPI is obtained using the equation:

$$CPI_A\ = \frac{\sum_i^n (CPI_i \times C_i)}{\sum_i^n C_i} = \frac{(1 \times 38) + (3 \times 15) + (4 \times 42) + (5 \times 5)}{38 + 15 + 42 + 5} = \frac{276}{100} = 2.76$$

So, the CPI of this machine A is 2.76.

4.5 CPU Execution Time

The basic performance equation in terms of *number of instructions executed by a program*, *CPI*, and *clock cycle time*:

$$CPU\ Time = instruction\ count \times\ CPI \times clock\ cycle\ time$$

Or, since the clock cycle time is the reciprocal of its frequency, the CPU time equation can be rewritten as:

$$CPU\ Time = \frac{instruction\ count \times\ CPI}{Clock\ rate}$$

4.6 Notions on CISC and RISC architectures

Current general processors are categorized into two broad categories, depending on the nature of the data set supported. The first category, called ***CISC*** (***Complex Instruction Set Computer***), is generally distinguished by a *large control* unit that occupies between 50% and 60% of the circuit area and a *microprogrammed sequencer*. The instructions of this architecture are complex, composed of several elementary operations. Given these drawbacks, a new architecture was developed (project initiated by IBM in 1975) and called ***RISC*** (***Reduced Instruction Set Computer***). In this architecture, the *control unit occupies less space*, and is equipped with a *hard-wired sequencer*.

4.6.1 Complex instruction Set Computer (CISC)

- Large number of complex instructions.
- Time-consuming instructions.
- Large number of addressing modes.
- Multiple instruction formats.

All of this requires a complex control unit to decode and execute the instructions.

Examples of processors of this type: VAX (DEC); 68xx (Motorola); x86, Pentium (Intel).

4.6.2 Reduced instruction Set Computer (RISC)

- Relatively small number of instructions.
- Small number of addressing modes.
- Small number of instruction formats.
- Instructions execute in a single cycle.
- Memory access is limited to read/write instructions.
- A large set of registers.

Examples of processors of this type: Alpha (DEC), PowerPC (Motorola), MIPS, SPARC.

Table 4.1 compares the two processors, VAX-11 and Berkeley RISC-1, which are based on CISC and RISC architectures, respectively.

Table 4.1: Comparison between VAX-11 (CISC) and Berkeley RISC-1 (RISC)

Characteristic	VAX-11 (CISC)	Berkeley RISC-1 (RISC)
Number of instructions	303	31
Instruction size (bits)	16-456	32
Addressing modes	22	3
Number of general-purpose registers	16	138

4.7 Presentation of the MIPS R3000 Microprocessor

The **MIPS R3000** (*Microprocessor without Interlocking Pipeline Stages*) is the second generation of processors from *MIPS Computer Systems*, announced in 1988. The MIPS R3000 is a **32-bit** industrial processor based on a **RISC** (*Reduced Instruction Set Computer*) instruction set, and it implemented **MIPS-I** instruction set architecture (ISA).

Several machines are built around the MIPS processor, including Sony and Nintendo gaming machines, many Cisco routers, TV set-top boxes, laser printers, and more.

4.8 External structure of the MIPS R3000 processor

The external architecture represents what a programmer wanting to code in assembler or someone wanting to develop a compiler for this processor needs to know:

- Visible registers.
- Memory addressing.
- The instruction set.
- Interrupt and exception handling mechanisms.

4.8.1 Registers

4.8.1.1 CPU Registers

The MIPS R3000 processor contains 35 user-visible registers, 32 of which are general-purpose (Table 4.2), and three special registers (***PC***, ***HI*** and ***LO***) whose use or modification is implicit with certain instructions. All registers are 32-bit in size.

Table 4.2: General-purpose registers of the MIPS R3000 processor

Register name	Register number	Use
Zero	0	Constant 0, writing does not change it
at	1	Reserved for assembler
v0, v1	2, 3	Used for function return values
a0, a1, a2, a3	4, 5, 6, 7	Used to pass arguments to functions
t0, t1…, t7	8, 9…, 15	Temporary (Caller-saved) Must be saved by the calling program.
s0, s1…, s7	16, 17…, 23	Temporarily save (Callee-saved). Must be saved by the called subroutine (Called Procedure)
t8, t9	24, 25	Temporary (Caller-saved) Must be saved by the calling program.
k0, k1	26, 27	Reserved for the operating system kernel
gp	28	Pointer to global variables
sp	29	Stack pointer
fp	30	Frame pointer
ra	31	Function call return address

- **Program Counter (PC):**
  This is a register initialized by the operating system. It contains the address of the next instruction to be executed. Once the instruction is loaded into the Instruction Register (IR), the PC value is incremented by **4 (because instructions are 4 bytes in size)**.
- **HI** and **LO:**
  These registers are used during division or multiplication. In the case of division, the **LO** register stores the quotient, while the **HI** register contains the remainder. In the case of a

multiplication, **LO** contains the 32 least significant bits and **HI** the 32 most significant bits (the result of the 64-bit multiplication).

4.8.1.2 System control co-processor (CP0) registers

The MIPS architecture has 32 registers, numbered 0 to 31, which are only accessible for reading and writing via privileged instructions. Privileged instructions are those that execute only in supervisor mode. These registers belong to the system control coprocessor (CP0).

In practice, four registers are used by this version of the MIPS R3000 processor for interrupt and exception handling (Table 4.3).

Table 4.3: MIPS R3000 processor registers used for interrupt and exception handling

Register name	Meaning of the letters	Register number	Use
BAR	Bad Address Register	8	In case of an exception of the type "illegal address", it stores the malformed address.
SR	Status Register	12	In particular, it contains the bit that determines the mode: supervisor or user, as well as the interrupt masking bits.
CR	Cause Register	13	When an interrupt or exception occurs, its contents define the cause for which the interrupt and exception handling program is called.
EPC	Exception Program Counter	14	In case of an interrupt, it stores the return address (PC + 4). In case of an exception, it contains the address of the faulty instruction (PC).

4.8.2 The Arithmetic and Logic Unit (ALU)

This unit performs arithmetic operations such as +, -, /, × and logical operations such as AND, OR, NOT, XOR. The operation is determined by an instruction.

4.8.3 The Control Unit (CU)

The control unit fetches, decodes, and executes instructions. To do this, it sends control signals to various components such as memory, registers, the ALU, etc. For example, the ALU needs a control signal to determine which operation it should perform.

The MIPS R3000 employs a classic **five-stage** RISC pipeline composed of instruction fetch (IF), instruction decode and register file access (ID), execution (EX), memory access (MEM), and write-back (WB).

4.8.4 Memory

The MIPS processor architecture specifies that a memory word contains 4 bytes (32 bits). The address of each memory location is also encoded in 32 bits (Figure 4.1). Since MIPS actually uses *byte addressing*, word addresses are multiples of 4.

In the MIPS architecture, addresses range from 0 to 2^{32}-1.

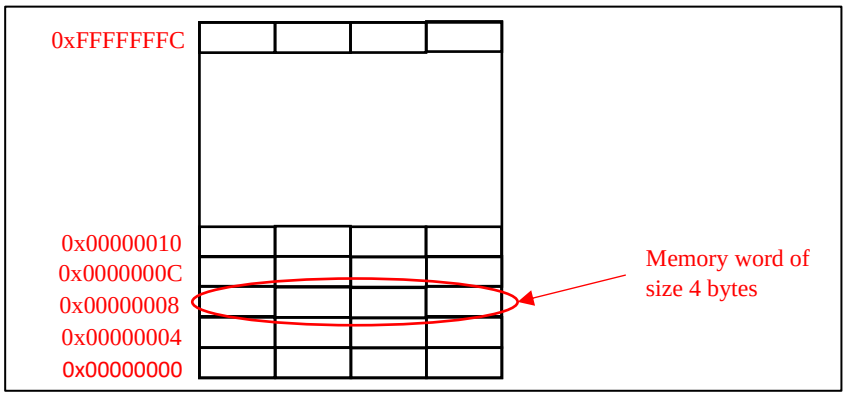


Figure 4.1: Illustrative diagram of a memory word

Regarding the way bytes are organized, MIPS operates, by default, in ***big-endian*** mode, however, it can also operate in ***little-endian*** mode (Figure 4.2).

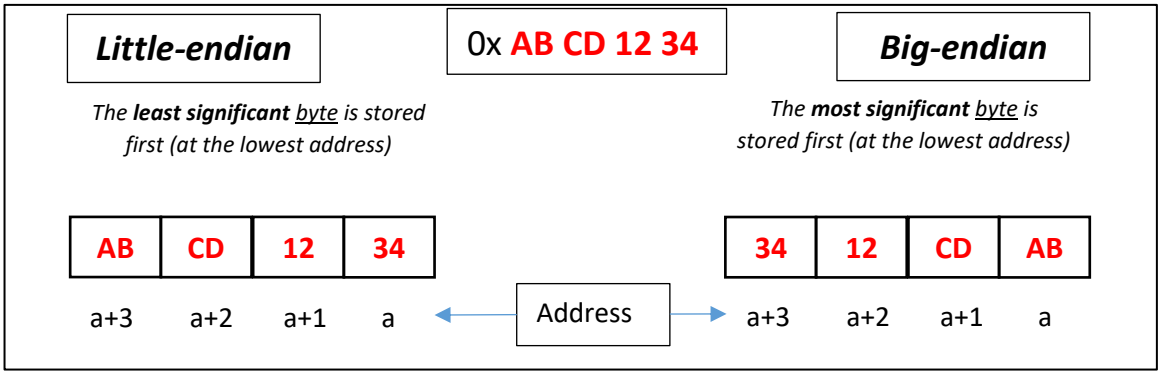


Figure 4.2 : Difference between little-endian and big-endian

4.8.4.1 General memory layout

The first portion of the lower end of memory is reserved and not available to user programs. Next is the Text section, where the MIPS machine code is stored. After this, we find the Static Data section, which contains constants and static variables. The Dynamic Data section contains data structures

whose size varies over their lifetime, such as linked lists. Finally, at the upper end of memory is the Stack. This contains the saved values of the argument registers, return addresses, etc.

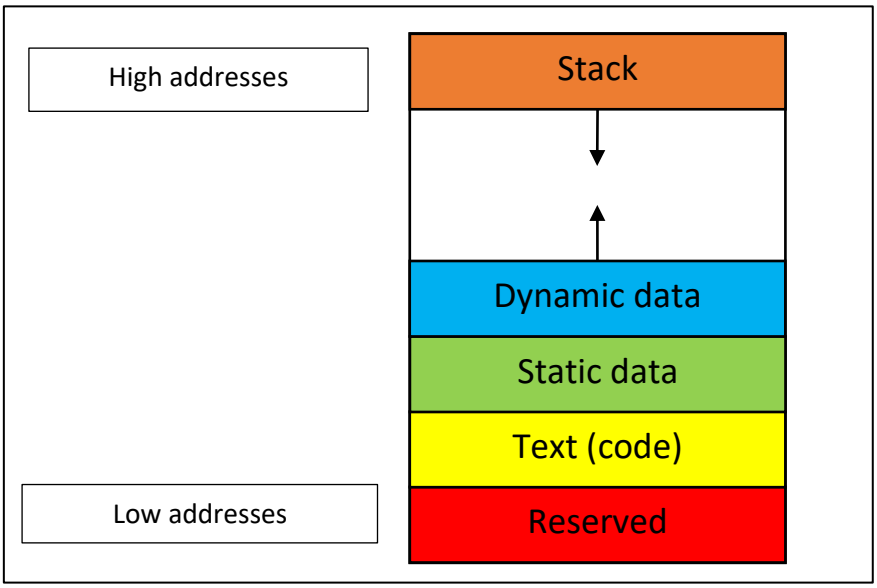


Figure 4.3: General memory layout for a program

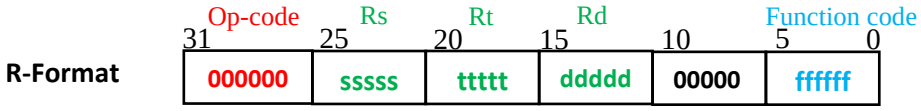
4.8.5 Instruction Register (IR)

The instruction register contains the currently executed instruction. It is 32 bits in size.

The MIPS architecture defines three instruction formats: ***R-Format*** (Register Format), ***I-Format*** (Immediate Format), and ***J-Format*** (Jump Format).

4.8.5.1 R-Format

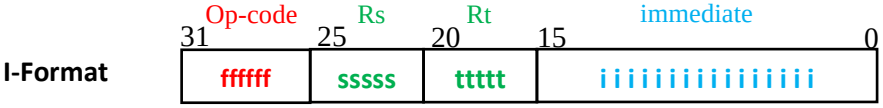
This format consists of 6 fields:



1. **Op-code (Operation code)**: Consists of 6 bits, all set to zero.
2. **Function code**: Consists of 6 bits. The function code specifies which function is to be executed.
3. **Rs**, **Rt** et **Rd**: Three fields specify the source registers (Rs and Rt) containing the operand values and the destination register (Rd) containing the result. Each register field is 5 bits in size.

4.8.5.2 I-Format

This format consists of 4 fields:



1. **Op-code (Operation code):** 6 bits long, specifies the operation to be performed by the processor.
2. **A source register (Rs)** containing the value of the first operand and a **destination register (Rt)** containing the result.
3. **Immediate**: 16 bits long, contains a constant representing the second operand.

4.8.5.3 J-Format

This format consists of two fields: the 6-bit **operation code** and the 26-bit **Target** field. The Target field specifies the target. This format is used by unconditional branch (**goto**) instructions

Figure 4.4 illustrates the basic functional components discussed in Section 4.8, including the CU, register file, ALU, memory, PC, and IR.

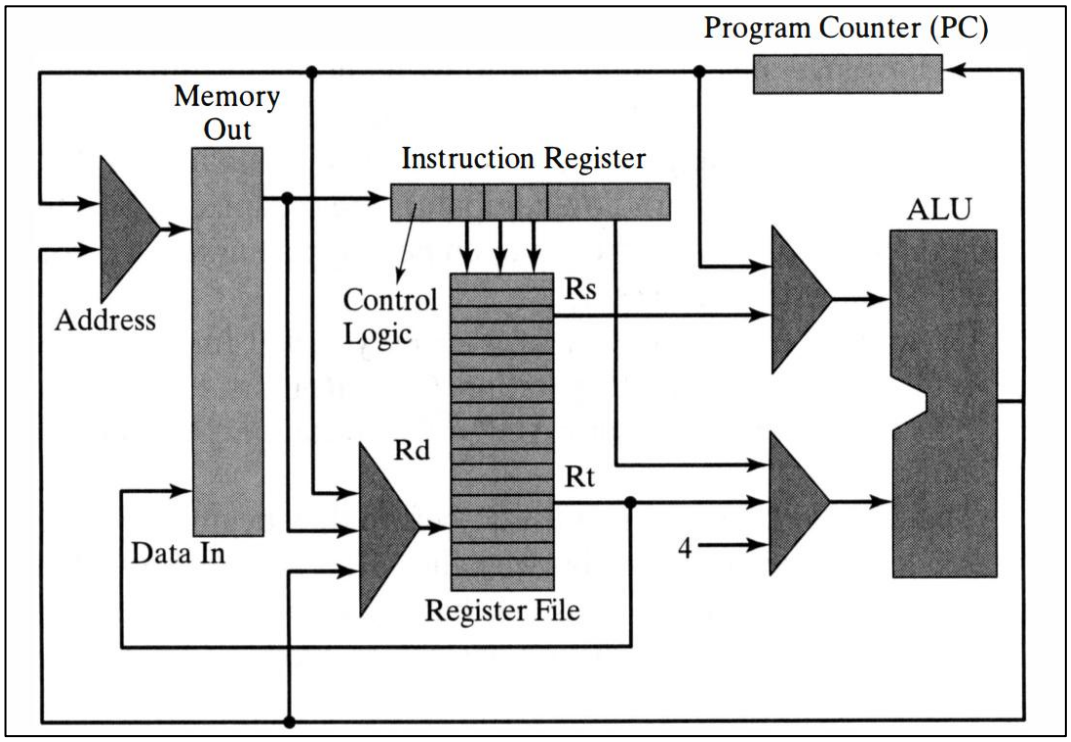


Figure 4.4: Simplified diagram of the MIPS data path [17]

Exercise 4.1: Using the MIPS simulator, test the code below and:

1. Assemble and run the program instruction by instruction (using 'Run one step at a time' button).
2. Identify which registers are used by the program.
3. Record the values of these registers after each instruction.

```
li $s0, 2017
li $s1, 2018
add $s3, $s0, $s1
```

4.9 Internal structure of the MIPS R3000 processor

4.9.1 Processor Interface

The interface between the processor and the memory is realized by the signals ADR[31:0], DATA[31:0], RW[2:0], FRZ, BERR (Figure 4.5). The possible requests to the memory are as follows:

RW	REQUETE	
000	NO	neither write nor read
001	WW	write a word
010	WH	write a half-word
011	WB	write a byte
1**	RW	read a word

In the **WW** case, the 4 bytes of the **DATA** bus are written to memory at a word-aligned address, ignoring the 2 least significant bits of **ADR**. In the **WH** case, the 2 least significant bytes of the **DATA** bus are written to a half-word-aligned address, ignoring the least significant bit of **ADR**. In the **WB** case, the least significant byte of the **DATA** bus is written to the address **ADR**. In the **RW** case, the two least significant bits of **ADR** are ignored, and the memory must provide a word-aligned **DATA** bus. In the case of **LBU** and **LHU** instructions, repositioning and sign extension are performed by the processor. If the memory system cannot satisfy the write or read request in one cycle (for example, in the event of a *cache miss*), the **FRZ** signal must be activated. As long as this signal is active, the processor maintains its request.

In the case of a hardware error while accessing memory, this error can be signaled to the processor by the **BERR** signal.

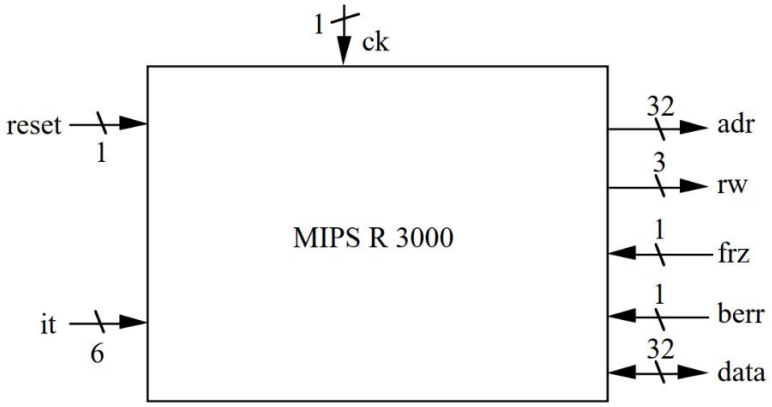


Figure 4.5: MIPS R3000 Processor Interface

4.9.2 Internal architecture of the processor

The processor's internal architecture consists of two parts: an **operative part** and a **control part**.

- The **operative part (OP)** groups the registers and operators. It performs elementary data transfers between one or more source registers and a destination register. An elementary transfer is executed in one cycle.

- The **control part (CP)** controls the operative part by defining, at each clock cycle, the elementary transfers to be executed (by the operative part). The control part primarily integrates a sequencer described as a finite state automaton (Moore automaton).

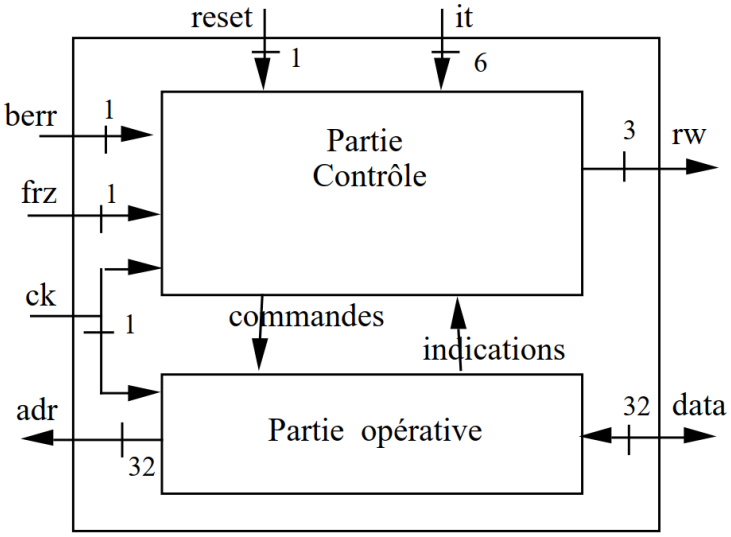


Figure 4.6: The internal architecture of the MIPS R3000 processor [20]

4.10 MIPS R3000 instruction set

The MIPS R3000 instruction set architecture (ISA) is organized into several classes of instructions, each designed to support a specific category of operations. These include arithmetic and logical instructions for integer computation (see section 4.10.2.1), data movement instructions for transferring data between registers and memory through load and store operations (see section 4.10.2.2), branch and jump instructions for control flow (see section 4.10.2.3), and system instructions for handling exceptions and system calls (see sections 5.3 and 5.4).

4.10.1 Encoding MIPS instructions

An instruction can primarily be identified by its operation code (**Op-Cod**), represented by bits **26** to **31** (Table 4.4). However, if this field contains one of the following codes: **000000 (SPECIAL)**, **000001 (BCOND)** or **010000 (COPRO)**, then other bits of the instruction must be checked to decode it (see Table 4.5, Table 4.6, Table 4.7, respectively).

For example, if the operation code is **000000 (SPECIAL)**, then the 6 least significant bits (0 to 5) must be analyzed.

Table 4.4: MIPS instruction encoding (OPCOD field: bits 31:26)

INS 31 :29 \ INS 28 :26	000	001	010	011	100	101	110	111
000	SPECIAL	BCOND	J	JAL	BEQ	BNE	BLEZ	BGTZ
001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
010	COPRO							
011								
100	LB	LH		LW	LBU	LHU		
101	SB	SH		SW				
110								
111								

Table 4.5 : MIPS instruction encoding (Function code field: bits 5:0)

OPCOD = SPECIAL

INS 5 :3 \ INS 2 :0	000	001	010	011	100	101	110	111
000	SLL		SRL	SRA	SLLV		SRLV	SRAV
001	JR	JALR			SYSCALL	BREAK		
010	MFHI	MTHI	MFLO	MTLO				
011	MULT	MULTU	DIV	DIVU				
100	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
101			SLT	SLTU				
110								
111								

Table 4.6: MIPS instruction encoding (OPCOD field = BCOND)

OPCOD = BCOND

INS 20 \ INS 16	0	1
0	BLTZ	BGEZ
1	BLTZAL	BGEZAL

Table 4.7: MIPS instruction encoding (OPCOD field = COPRO)

OPCOD = COPRO

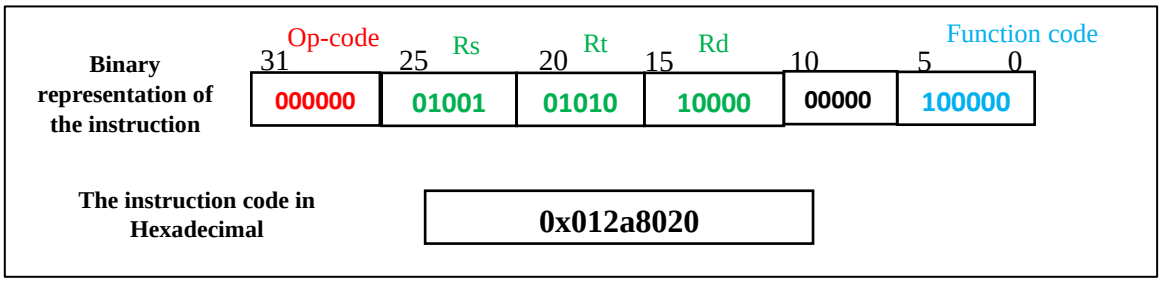
INS 25 \ INS 23	0	1
0	MFC0	MTC0
1	RFE	

Example of instruction encoding

Consider the instruction **add $s0, $t1, $t2**:

- ✓ This instruction is clearly in ***R-format***.
- ✓ According to the previous tables:
  - The function code (bits 5 to 0) is: **100 000**
  - The operation code is ***Special***, so bits 31 to 26 are **000 000**
- ✓ The registers involved:
  - The first source register (Rs): **$t1** (or **$9**) represented in binary on 5 bits: **01001**
  - The second source register (Rt): **$t2** (or **$10**) represented in binary on 5 bits: **01010**
  - The destination register (Rd): **$s0** (or **$16**) represented in binary on 5 bits: **10000**

This gives us the following binary representation:



4.10.2 MIPS Instruction Set

4.10.2.1 Arithmetic and Logic Instructions

1. **Addition** register register signed
   (add)
   Syntax: **add $rd, $rs, $rt**
   Operation: $rd ← $rs + $rt
2. **Addition** register register unsigned
   (add unsigned)
   Syntax: **addu $rd, $rs, $rt**
   Operation: $rd ← $rs + $rt
3. **Addition** register immediate signed
   (add Immediate)
   Syntax: **addi $rt, $rs, imm**
   Operation: $rt ← imm_{15}^{16} || $imm_{15\ldots0}$ + $rs
4. **Addition** register immediate unsigned
   (add Immediate unsigned)
   Syntax: **addiu $rt, $rs, imm**
   Operation: $rt ← imm_{15}^{16} || $imm_{15\ldots0}$ + $rs
5. **Subtraction** register register signed
   (subtract)
   Syntax: **sub $rd, $rs, $rt**
   Operation: $rd ← $rs - $rt
6. **Subtraction** register register unsigned
   (subtract unsigned)
   Syntax: **subu $rd, $rs, $rt**
   Operation: $rd ← $rs - $rt
7. **Multiplication** register register signed
   (multiply)
   Syntax: **mult $rs, $rt**
   Operation: $lo ← $(\$rs \times \$rt)_{31\ldots0}$
   $hi ← $(\$rs \times \$rt)_{63\ldots32}$
8. **Multiplication** register register unsigned
   (multiply unsigned)
   Syntax: **multu $rs, $rt**
   Operation: $lo ← $(0||\$rs \times 0||\$rt)_{31\ldots0}$
   $hi ← $(0||\$rs \times 0||\$rt)_{63\ldots32}$
9. **Division and remainder signed** register register
   (divide)
   Syntax: **div $rs, $rt**
   Operation: $lo ← $\frac{\$rs}{\$rt}$
   $hi ← $rs mod $rt
10. **Division and remainder unsigned** register register
    (divide unsigned)
    Syntax: **divu $rs, $rt**
    Operation: $lo ← $\frac{0||\$rs}{0||\$rt}$
    $hi ← 0||$rs mod 0||$rt

11. **Bitwise logical and** register register
(and)
Syntax: **and $rd, $rs, $rt**
Operation: $rd ← $rs and $rt
12. **Bitwise logical and** register immediate
(and Immediate)
Syntax: **andi $rt, $rs, imm**
Operation: $rt ← ($0^{16}$|| imm) and $rs
13. **Bitwise or** register register
(or)
Syntax: **or $rd, $rs, $rt**
Operation: $rd ← $rs or $rt
14. **Bitwise or** register immediate
(or Immediate)
Syntax: **ori $rt, $rs, imm**
Operation: $rt ← ($0^{16}$|| imm) or $rs
15. **Bitwise Exclusive-Or** Register Register
(Exclusive or)
Syntax: **xor $rd, $rs, $rt**
Operation: $rd ← $rs xor $rt
16. **Bitwise Exclusive-Or** Register immediate
(Exclusive or Immediate)
Syntax: **xori $rt, $rs, imm**
Operation: $rt ← ($0^{16}$|| imm) xor $rs
17. **Bitwise Not-Or** Register Register
(nor)
Syntax: **nor $rd, $rs, $rt**
Operation: $rd ← $rs nor $rt
18. **mfhi** copies the contents of the $Hi register into a general register.
(Move from Hi)
Syntax: **mfhi $rd**
Operation: $rd ← $hi
19. **mflo** copies the contents of the $Lo register into a general register.
(Move from Lo)
Syntax: **mflo $rd**
Operation: $rd ← $lo
20. **mthi** copies the contents of a general register into $Hi.
(Move to Hi)
Syntax: **mthi $rs**
Operation: $hi ← $rs
21. **mtlo** copies the contents of a general register into $Lo.
(Move to Lo)
Syntax: **mtlo $rs**
Operation: $lo ← $rs
22. **load a 32-bit immediate into register (pseudo-instruction)**
(Load Immediate)
Syntax: **li $rt, imm**
Operation: $rt ← imm

Exercise 4.2:

1- Write an assembler program (MIPS) that loads two values, **1988** and **2018,** into two registers (e.g., **$s0** and **$s1**), then performs **addition**, **subtraction**, **AND**, **OR**, and **XOR**. The results are stored in the **t_i$** registers.
2- Write an assembler program (MIPS) that loads two values, **2018** and **1988**, into two registers (e.g., **$s0** and **$s1**), then performs **multiplication** and **division**.

What instructions should be used to store the results in the general registers?
Note: It is possible to use the instruction "**mul $rd, $rs, $rt**".

4.10.2.2 Data transfer instructions (read/write Memory)

1. **Writing a byte to memory**
   (store byte)
   Syntax: **sb $rt, imm($rs)**
   Operation: mém[$rs+imm] ← rt_{7...0}$
2. **Writing a half-word into memory**
   (store half word)
   Syntax: **sh $rt, imm($rs)**
   Operation: mém[$rs+imm] ← rt_{15...0}$
3. **Writing a word from memory**
   (store word)
   Syntax: **sw $rt, imm($rs)**
   Operation: mém[$rs+imm] ← $rt
4. **Reading a signed byte from memory**
   (load byte)
   Syntax: **lb $rt, imm($rs)**
   Operation: $rt ← mém[$rs+imm]$_7^{24}$ || mém[$rs+imm]$_{7...0}$
5. **Reading an unsigned byte from memory**
   (load byte unsigned)
   Syntax: **lbu $rt, imm($rs)**
   Operation: $rt ← 0^{24} || mém[$rs+imm]$_{7...0}$
6. **Reading a signed half-word from memory**
   (load half word)
   Syntax: **lh $rt, imm($rs)**
   Operation: $rt ← mém[$rs+imm]$_{15}^{16}$ || mém[$rs+imm]$_{15...0}$
7. **Reading an unsigned half-word from memory**
   (load half word unsigned)
   Syntax: **lhu $rt, imm($rs)**
   Operation: $rt ← 0^{16} || mém[$rs+imm]$_{15...0}$
8. **Reading a word from memory**
   (load word)
   Syntax: **lw $rt, imm($rs)**
   Operation: $rt ← mém[$rs+imm]

Exercise 4.3:

Let the value $(1234567890)_{10}$ = $(100\ 1001\ 1001\ 0110\ 0000\ 0010\ 1101\ 0010)_2$

1. What are values stored if we use the following instructions:
   li $s1, 1234567890
   li $t1, 0x10010000
   sb $s1, ($t1)
   li $t2, 0x10010004
   sh $s1, ($t2)
   li $t3, 0x10010008
   sw $s1, ($t3)
2. What are the memory locations used to store these values?
3. Verify your solution by implementing this code on MARS.
4. Modify the code so that a single base address is used.

Exercise 4.4:

From memory location **0x10010008** (from the previous exercise):

1. Load a single byte, half word and the memory word into registers $s2, $s3 and $s4 respectively.
2. Discuss the different instructions.
   lb $s2, ($t3)
   lh $s3, ($t3)
   lw $s4, ($t3)

4.10.2.3 Branch instructions

1. **Branch if register equals register**
   (branch if equal)
   Syntax: **beq $rs, $rt, imm**
   Operation: if (($rs) == ($rt)) then
   $PC=$PC+(imm_{15}^{14} || $imm_{15...0}$|| 0^2)
2. **Brach if register different from register**
   (branch if not equal)
   Syntax: **bne $rs, $rt, imm**
   Operation: if (($rs) != ($rt)) then
   $PC=$PC+(imm_{15}^{14} || $imm_{15...0}$|| 0^2)
3. **Branch if register greater than or equal to zero**
   (branch if greater or equal zero)
   Syntax: **bgez $rs, imm**
   Operation: if (($rs)>=($zero)) then
   $PC=$PC+(imm_{15}^{14} || $imm_{15...0}$|| 0^2)
4. **Branch if register strictly greater than zero**
   (branch if greater than zero)
   Syntax: **bgtz $rs, imm**
   Operation: if (($rs) > ($zero)) then
   $PC=$PC+(imm_{15}^{14} || $imm_{15...0}$|| 0^2)
5. **Branch if register less than or equal to zero**
   (branch if less or equal zero)
   Syntax: **blez $rs, imm**
   Operation: if (($rs) <= ($zero)) then
   $PC=$PC+(imm_{15}^{14} || $imm_{15...0}$|| 0^2)
6. **Branch if register strictly less than zero**
   (branch if less than zero)
   Syntax: **bltz $rs, imm**
   Operation: if (($rs) < ($zero)) then
   $PC=$PC+(imm_{15}^{14} || $imm_{15...0}$|| 0^2)
7. **Branch to a function if register greater than or equal to zero**
   (branch if greater or equal zero and link)
   Syntax: **bgezal $rs, imm**
   Operation: if (($rs) >= ($zero)) then
   $ra=$PC
   $PC=$PC+(imm_{15}^{14} || $imm_{15...0}$|| 0^2)
8. **Branch to a function if register strictly less than zero**
   (branch if less than zero and link)
   Syntax: **bltzal $rs, imm**
   Operation: if (($rs) < ($zero)) then
   $ra=$PC
   $PC=$PC+(imm_{15}^{14} || $imm_{15...0}$|| 0^2)
9. **Unconditional branch immediate**
   (jump)
   Syntax: **j imm**
   Operation: $PC=$PC_{31...28}$ || $imm_{25...0}$|| 0^2
10. **Unconditional function call immediate**
    (jump and link)
    Syntax: **jal imm**
    Operation: $ra=$PC
    $PC=$PC_{31...28}$ || $imm_{25...0}$|| 0^2
11. **Unconditional branch register**
    (jump register)
    Syntax: **jr $rs**
    Operation:
    $PC=$rs
12. **Unconditional function call register**
    (jump and link register)
    Syntax: **jalr $rs**
    Operation: $ra=$PC
    $PC= $rs
13. **Unconditional function call register**
    (jump and link register)
    Syntax: **jalr $rd, $rs**
    Operation: $rd=$PC
    $PC= $rs

Exercise 4.5:

1. Write a program in MIPS assembly language that loads two integers into two registers (**$s0** and **$s1**), then places the number of the register containing the larger value into the register **$t0**.
   Note: If the values are equal, register $t0 is set to -1.
2. Write a program that calculates the **absolute value** of a variable X.
   Note: Assuming the value of variable X is loaded into the register $s0, the result (i.e., the absolute value) is assigned to register $s1.

4.11 MIPS Processor Addressing Modes

Five addressing modes are available with the MIPS processor:

a) ***Immediate addressing***: The operand is embedded in the instruction itself as a constant.
b) ***Register addressing***: The operand is located in a register.
c) ***Base or displacement addressing***: The operand is located in a memory location whose address is obtained by adding a displacement to a base address. The address is located in a register, and the displacement is represented by a constant in the instruction
d) ***PC-relative addressing***: The branch address is the sum of the PC and a constant in the instruction.
e) ***Pseudo-direct addressing***: The jump address is the concatenation of the 26 bits in the instruction with the most significant bits of the PC.

These five addressing modes are illustrated in Figure 4.7 below.

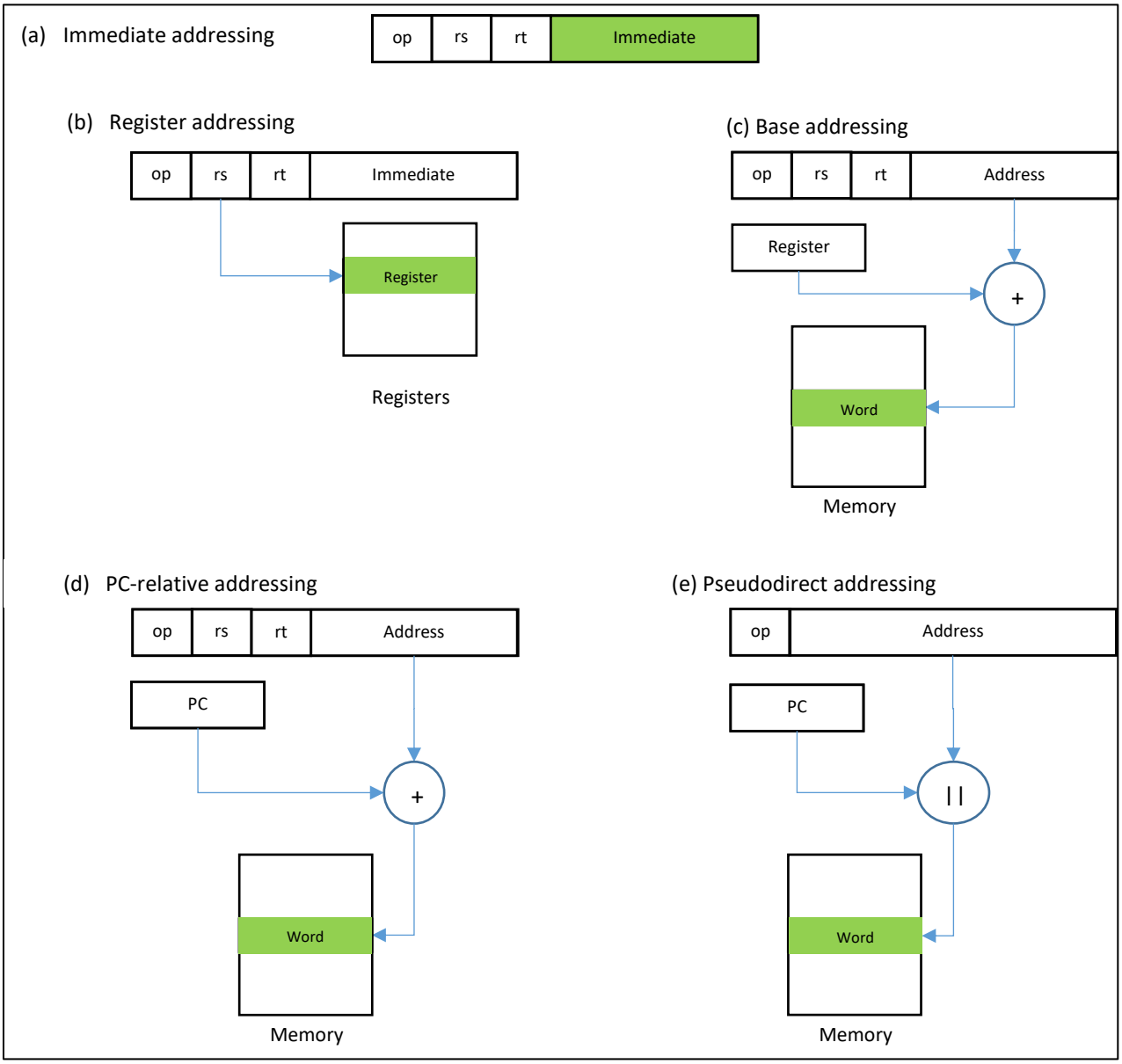


Figure 4.7: Illustration of the five addressing modes of the MIPS processor

4.12 Programming in MIPS Assembly Language

In this section, we will see through examples how to translate some control structures in the form of pseudo-codes in MIPS assembly language. We will also see in a second point the assembler directives.

4.12.1 Translation of some control structures

In this section, one conditional structure (***if…then…else…***) and two loop structures (***while*** and ***for***) are translated from pseudo-code to MIPS assembly languages.

4.12.1.1 If … then …else…

Example: Perform the translation of the following conditional structure.

```
if ($t8 < 0) then {
      $s0 = 0 - $t8
      $tl = $tl + 1}
else {
      $s0 = $t8
      $t2 = $t2 + l}
```

Solution:

```
      bgez $t8, else
      sub $s0, $zero, $t8
      addi $t1, $t1, 1
      j next
else: move $s0, $t8
      addi $t2, $t2, 1
next:
```

4.12.1.2 While

Example: Perform the translation of the loop structure.

```
while ($a1 < $a2) do {
      $a1 = $a1 + 1
      $a2 = $a2 - 1}
```

Solution:

```
while: bge $a1, $a2, done
       addi $a1, $a1, 1
       addi $a2, $a2, -1
       j while
done:
```

Test: Rewrite the “while” loop using the “bgez” instruction.

4.12.1.3 For

Example: Perform the translation of the “for” loop structure.

```
$a0 = 0;
for ($t0= 10; $t0> 0; $t0= $t0 - 1 ) do{
$a0 = $a0 + $t0}
```

Solution:

```
li $a0 , 0
li $t0, 10
loop: blez $t0 , done
      add $a0 , $a0 , $t0
      addi $t0 , $t0, -1
      j loop
done:
```

4.12.2 Assembler Directives

An assembler directive is a message sent to the assembler to tell it something it needs to know to complete the assembly process. The MIPS assembly language has several directives. These are identified by the fact that they begin with the dot symbol ".". Directives allow the programmer to:

- Specify the memory segment;
- Describe visibility;
- Describe and store data.

The Table 4.8 presents some directives.

Table 4.8: Some assembler directives

Directives	Explanation
.data <addr>	Data must be declared in this section. If the optional addr argument is provided, the following elements are stored starting at address addr.
.text <addr>	The code must be placed in the text section. If the optional addr argument is provided, the following elements are stored starting from the addr address.
.byte b1, …, bn	n values of size 8 bits for each are stored in successive bytes of memory.
.half h1, …, hn	n values of size 16 bits for each are stored in successive half-words of memory.
.word w1, …, wn	n values of size 32 bits for each are stored in successive words of memory.
.ascii string	Store the string in memory, but do not terminate it with a null character.
.asciiz string	Store the string in memory and terminate it with null.
.float f1, …, fn	n single-precision floating-point numbers are stored in successive memory locations.
.double d1, …, dn	n double-precision floating-point numbers are stored in successive memory locations.
.space n	Allocate n bytes of space in the current segment.

4.12.2.1 Structure of a MIPS assembly language program

In the following, we will use both ***.data*** and ***.text*** directives to structure our program (Figure 4.8). As shown earlier in Table 4.8, data is declared in the ***.data*** section, while executable instructions are written in the ***.text*** section.

```
1  # Name and general description of the program
2  # ---------------------------------------
3  # The data declaration section (.data)
4  .data
5  # Here we declare the program-specific data.
6
7  # ---------------------------------------
8  # The program code section (.text)
9  .text
10 # Here we write the program code.
11
12 # -----
```

Figure 4.8: .data and .text sections of an assembly program

An assembly program can also contain comments and labels:

- ✓ **Labels**: designate locations in the code, often used as function/procedure names or as jump destination points.
  The rules for a label are as follows:
  - It must begin with a letter.
  - It can be followed by letters, numbers, or an "_" (underscore).
  - It must end with a ":" (colon).
  - It can only be defined once.
- ✓ **Comments**: The "#" character is used to indicate a comment line. Therefore, anything written after the "#" is considered a comment.

Therefore, the general syntax of a MIPS assembly language instruction is as follows:

```
[label:] opCode [operand], [operand], [operand] [# comment]
```

Here, square brackets are used to indicate optional fields. These fields depend on the operation.

4.12.2.2 Data declaration

Data declaration is done in the ***.data*** section using the following general syntax:

```
<variableName>: .<dataType> <initialValue>
```

The variable name is followed by a ":", then the data type (see the data declaration directives in the previous table), and finally the initial value assigned to the variable.

A. Integer Declaration:

Integer declarations are performed using the following directives: ***.word***, ***.half***, ***.byte***.

Examples:

```
wEntier1: .word 202409
wEntier2: .word -240927
hEntier1: .half 4321
hEntier2: .half -4000
bEntier1: .byte 35
bEntier2: .byte -24
```

Notes:

- Two's complement is used to represent integers.
- Initializing a variable to a value that cannot be stored in the allocated space generates an assembly error (*MARS truncates the binary representation*).

B. Declaring Strings:

A string is a series of byte-sized characters defined sequentially, usually terminated by a NULL byte (0x00). String declarations are accomplished using the following directives: ***.ascii***, ***.asciiz***.

Example:

```
message: .asciiz "Bonjour monde\n"
```

Notes:

- The ASCII standard is used to represent characters.
- Newline ("\n") and tab ("\t") sequences, similar to C/C++, are supported in strings.

C. Declaration of Floating-Point Numbers:

Floating-point numbers are defined by the directives: ***.float*** (32 bits), ***.double*** (64 bits).

Examples:

```
pi: .float 3.14159
e: .double 2.718281
```

Notes:

- The IEEE standard is used to represent floating-point numbers.

Figure 4.9 illustrates the data declaration examples provided above.

```
1  # Name and general description of the program
2  # ----------------------------------------
3  # The data declaration section (.data)
4  .data
5  # Here we declare the program-specific data.
6  message: .asciiz "Bonjour monde\n"
7  wEntier1: .word 202409
8  wEntier2: .word -240927
9  hEntier1: .half 4321
10 hEntier2: .half -4000
11 bEntier1: .byte 35
12 bEntier2: .byte -24
13 pi: .float 3.14159
14 e: .double 2.718281
15 # ----------------------------------------
16 # The program code section (.text)
17 .text
18 # Here we write the program code.
19
20 # -----
```

Figure 4.9: Example of declarations of some data of different types

Chapter 5. Special Instructions

5.1 CONCEPTS ABOUT INTERRUPTIONS AND EXCEPTIONS 82
5.1.1 Exceptions 82
5.1.2 Interrupts 83
5.1.3 System calls: SYSCALL and BREAK instructions 83
5.1.4 RESET signal 83
5.2 INPUT/OUTPUT 84
5.2.1 Input 84
5.2.2 Output 87
5.2.3 Terminating a MIPS Program Properly 89
5.3 SPECIAL INSTRUCTIONS 92
5.4 SYSTEM CONTROL COPROCESSOR (CP0) INSTRUCTIONS 92

5.1 Concepts about interruptions and exceptions

Interrupts and exceptions are events that modify the normal flow of instruction execution. In fact, the interrupt is generated by a signal from the hardware (from outside the processor), the time of occurrence of the interrupt is independent of the execution of the program. In other words, the interrupt is asynchronous. Most often, it is caused by the input/output devices. On the other hand, the exception is caused by the execution of an instruction and is used to, for example, signal an overflow or division by zero. The exception is synchronous with the execution of the program.

There are four types of events that can interrupt the normal execution of a program on the MIPS R3000 processor:

- Exceptions
- Interrupts
- System calls (SYSCALL and BREAK instructions)
- The RESET signal

When a program is interrupted by one of the above events, control is transferred to a specialized software procedure, which runs in supervisor mode and must be provided with the minimum information needed to handle the problem.

5.1.1 Exceptions

Exceptions are "abnormal" events, usually caused by programming errors, that prevent the current instruction from being executed correctly. When an exception is detected, the execution of the faulty instruction is immediately interrupted. This ensures that the faulty instruction does not alter the value of a register or memory. Exceptions are obviously not maskable. This version[10] of the R3000 processor recognizes 7 types of exceptions (Table 5.1).

Table 5.1: Seven types of exceptions recognized by the MIPS R3000 processor

ADEL	**Address Error Load**	Unaligned address or address located in the system segment while the processor is in user mode.
ADES	**Address Error Store**	Unaligned address or access to data in the system segment while the processor is in user mode.
DBE	**Data Bus Error**	The memory system signals an error by activating the BERR signal following a data access.
IBE	**Instruction Bus Error**	The memory system signals an error by activating the BERR signal during an instruction read.
OVF	**Overflow**	When executing an arithmetic instruction (ADD, ADDI, or SUB), the result cannot be represented in 32 bits.
RI	**Reserved Instruction**	The codop does not correspond to any known instruction (this is probably a branch to a memory area that does not contain executable code).
CPU	**Coprocesseur inaccessible**	An attempt to execute a privileged instruction (MTC0, MFC0, RFE) while the processor is in user mode.

In the event of an exception, the processor enters supervisor mode and executes the exception handler located at address "0x80000080". Since all exceptions are fatal on this R3000 processor, no recovery

[10] *Processeur MIPS R3000. Architecture externe*, UNIVERSITE PIERRE ET MARIE CURIE, 2003

of the faulty program is provided, making saving the return address unnecessary. However, the processor transmits the address of the faulty instruction and the exception type to the handler via the cause register.

When an exception is detected, the processor:

- Saves the address of the faulty instruction in the **EPC** register
- Saves the old value in the **SR** status register
- Enters supervisor mode and masks interrupts in **SR**
- Writes the exception type to the **CR** register
- Branches to address "0x80000080".

5.1.2 Interrupts

Hardware interrupts, generated by external devices, are asynchronous events that can be masked. The processor has six external interrupt lines that can be masked globally or individually. Activating one of these lines constitutes an interrupt request. They are unconditionally written to the CR register and, if not masked, are processed at the end of the current instruction. The device must keep the request active until it is processed. The processor then enters supervisor mode, accesses the exception handler, and saves the return address to resume program execution after processing.

When an unmasked interrupt request is detected, the processor:

- Saves the return address (PC + 4) in the **EPC** register
- Saves the old value in the **SR** status register
- Enters supervisor mode and masks interrupts in **SR**
- Writes that it is an interrupt to the **CR** register
- Branches to address "0x80000080".

The R3000 processor integrates a software interrupt mechanism, in addition to the six hardware interrupt lines. Two bits in the cause register (CR) can be enabled by the privileged MTC0 instruction. Setting these bits to 1 triggers the same processing as external interrupt requests if they are not masked.

5.1.3 System calls: SYSCALL and BREAK instructions

The SYSCALL instruction allows a task to request an operating system service, such as an input/output operation, by providing the service code and any parameters in general-purpose registers. The BREAK instruction is used more specifically to set a breakpoint (for software debugging purposes): an instruction in the program to be debugged is abruptly replaced with the BREAK instruction. In both cases, the processor enters supervisor mode and accesses the exception handler. These two instructions are executable in user mode. They perform the following operations:

- Save the return address (PC + 4) in the **EPC** register
- Save the old value in the **SR** status register
- Switch to supervisor mode and mask interrupts in **SR**
- Write the cause of the trap to the **CR** register
- Branch to address "0x80000080".

5.1.4 RESET signal

The processor has a RESET line which, when enabled for at least one cycle, triggers an unconditional branch to the initialization software. This request is very similar to a seventh external interrupt line with the following important differences:

- ❖ It is not maskable.
- ❖ It is not necessary to save a return address.
- ❖ The reset handler is located at address "0xBFC00000".

In this case, the processor:

- Enters supervisor mode and masks interrupts in **SR**
- Branches to address "0xBFC00000".

5.2 Input/Output

Several system services are available for use in MIPS-based programs. The codes for 10 services[11] are shown in Table 5.2 below.

To request a specific service, follow these steps:

1. Load the service number into the register $v0.
2. Load the argument values, if any, into $a0, $a1, $a2, or $f12 as specified.
3. Execute the SYSCALL instruction.
4. Retrieve the return values, if any, from the result registers as specified.

Table 5.2: the top ten MIPS assembly language services and their codes

Service	Code in $v0	Arguments	Result
print integer	1	$a0 = integer to display	
print float	2	$f12 = float to display	
print double	3	$f12 = double to display	
print string	4	$a0 = Address of the null-terminated string to display	
read integer	5		$v0 contains the integer read
read float	6		$f0 contains the real number (float) read
read double	7		$f0 contains the real number (double) read
read string	8	$a0 = address of the input buffer $a1 = maximum number of characters to read	
sbrk (allocate heap memory)	9	$a0 = number of bytes to allocate	$v0 contains the address of the allocated memory
exit (terminate execution)	10		

5.2.1 Input

In this paragraph we will see examples of reading values entered by the user.

[11] For the exhaustive list of services available with the MARS simulator, you can consult the simulator help or its website.

A. <u>Reading an integer:</u>

To do this, we need to load $v0 using the integer reading code, which is 5 (see Table 5.2). Since we don't need any arguments, we execute the **syscall** instruction directly. Once the integer is entered by the user, it will be stored in the $v0 register (Figure 5.1).

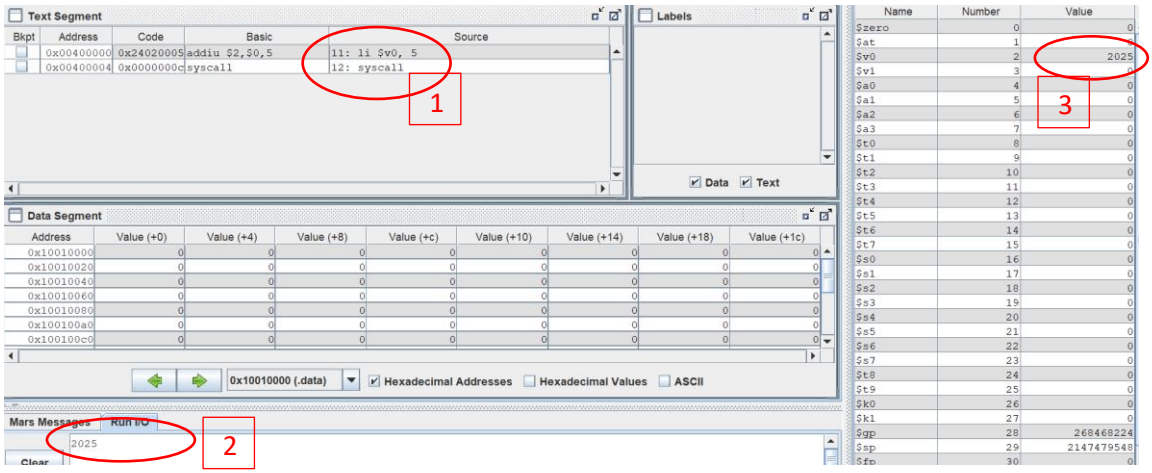


Figure 5.1: Reading an integer

B. <u>Reading a real number:</u>

For this purpose, we need to load the $v0 register with the real number reading code, which is 6 for single precision (float) (in case the number is double precision (double) the code is 7, see Table 5.2). Since we don't need any argument, we execute the **syscall** instruction directly. Once the number is entered by the user, it will be stored in the $f0 register (Figure 5.2).

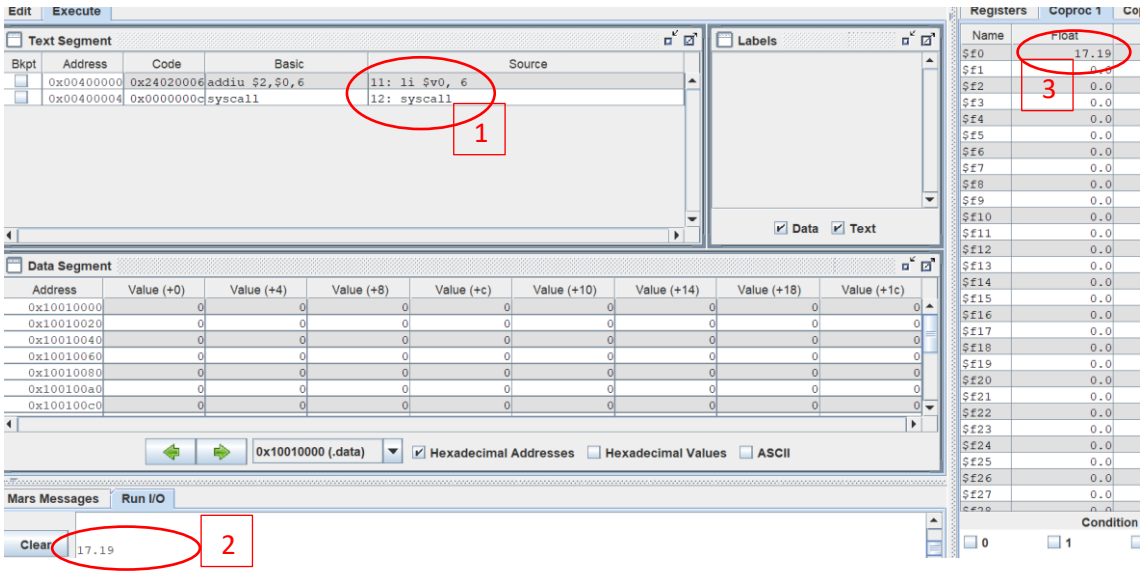


Figure 5.2: Reading a real number

C. Reading a string:

To read a string, the $v0 register must be loaded with the value 8 (see Table 5.2). Here, we need two arguments; the first is the address of the input buffer where the string will be stored (in memory), and the second is the maximum size of the string. Then, we execute the **syscall** instruction. Once the string is entered by the user, it will be stored in memory starting from the buffer address (Figure 5.4 and Figure 5.3).

ReadingString.asm

```
1  # String reading program
2  # ---------------------------------------
3  # The data declaration section (.data)
4  .data
5  # Here we declare the program-specific data.
6  userInput:  .space 20
7  # ---------------------------------------
8  # The program code section (.text)
9  .text
10 # Here we write the program code.
11 li $v0, 8
12 la $a0, userInput
13 li $a1, 20
14 syscall
15 # -----
```

Figure 5.4: Code for reading a character string

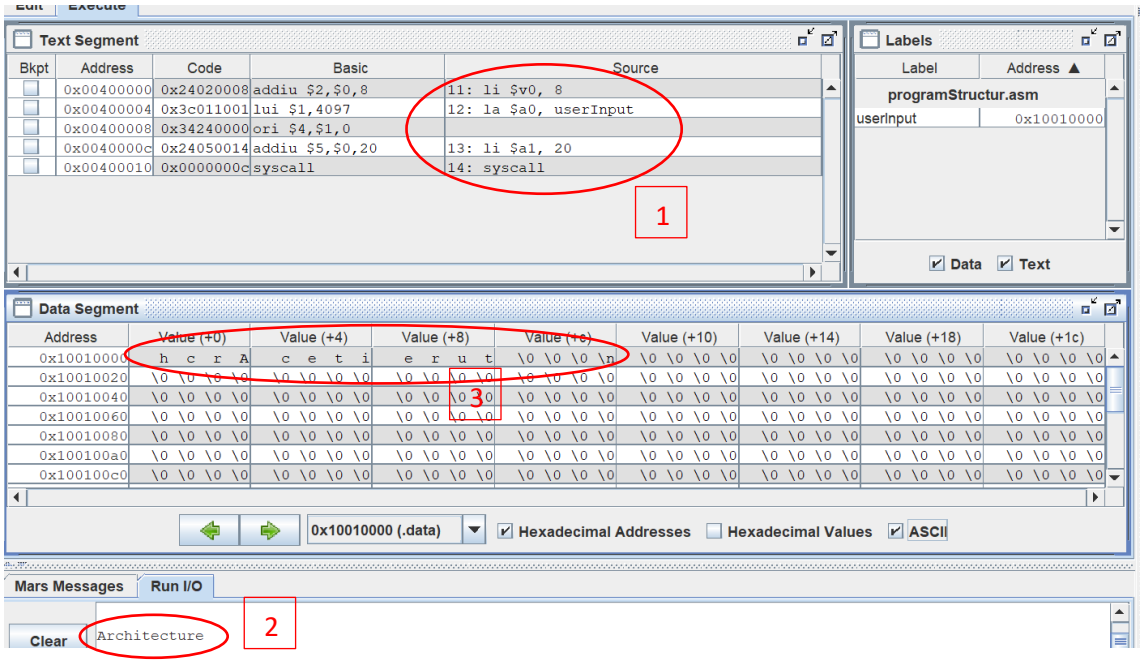


Figure 5.3: Reading a string

5.2.2 Output

Now, we'll give some examples of printing data of different types (integer, real, string).

A. <u>Printing an integer:</u>

To print an integer, the register $v0 must be loaded by the corresponding code, i.e., 1. Then in a second step, the register $a0 is loaded with the integer to be printed. Finally, the **syscall** instruction is executed (Figure 5.5).

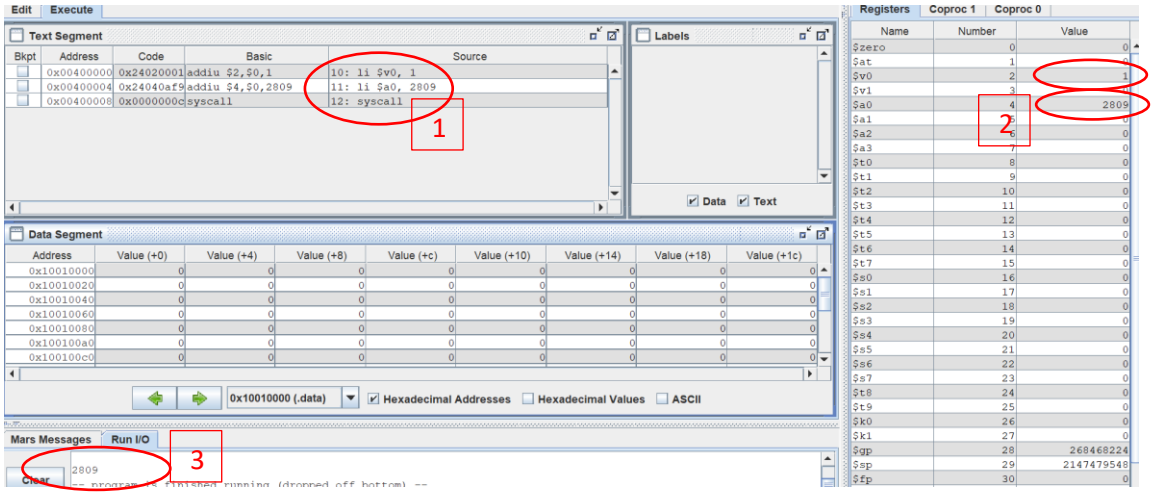


Figure 5.5: Printing an integer

B. <u>Printing a real number:</u>

To do this, we need to load the $v0 register with the real number print code: 2 for single precision (***float***) and 3 for double precision (***double***). The real number to be printed must be loaded into $f12, then the **syscall** instruction is executed (Figure 5.6 and Figure 5.7).

printFloat.asm

```
1   .data
2   # Here we declare the program-specific data.
3   e: .float 2.718281
4   # ----------------------------------------
5   # The program code section (.text)
6   .text
7   # Here we write the program code.
8   li $v0, 2
9   lwc1 $f12, e
10  syscall
11  # -----
```

Figure 5.6: Code for printing a real number

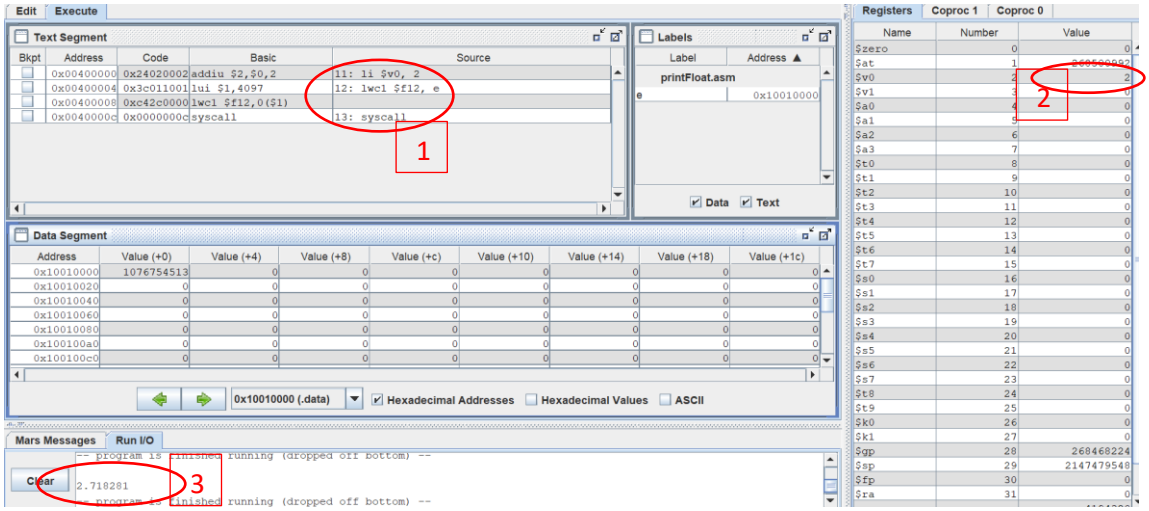


Figure 5.7: Printing a real number

C. <u>Printing a String:</u>

To do this, we need to load the $v0 register with the string print code, which is 4. Then we load the address of the string to be printed into the $a0 register. Finally, we execute the **syscall** instruction (Figure 5.8 and Figure 5.9).

Edit Execute

printString

```
1  # String printing program
2  # ----------------------------------------
3  # The data declaration section (.data)
4  .data
5  # Here we declare the program-specific data.
6  msg: .asciiz "Architecture des Ordinateurs\n"
7  # ----------------------------------------
8  # The program code section (.text)
9  .text
10 # Here we write the program code.
11 li $v0, 4
12 la $a0, msg
13 syscall
14 # -----
```

Figure 5.8: Code for printing a string of characters

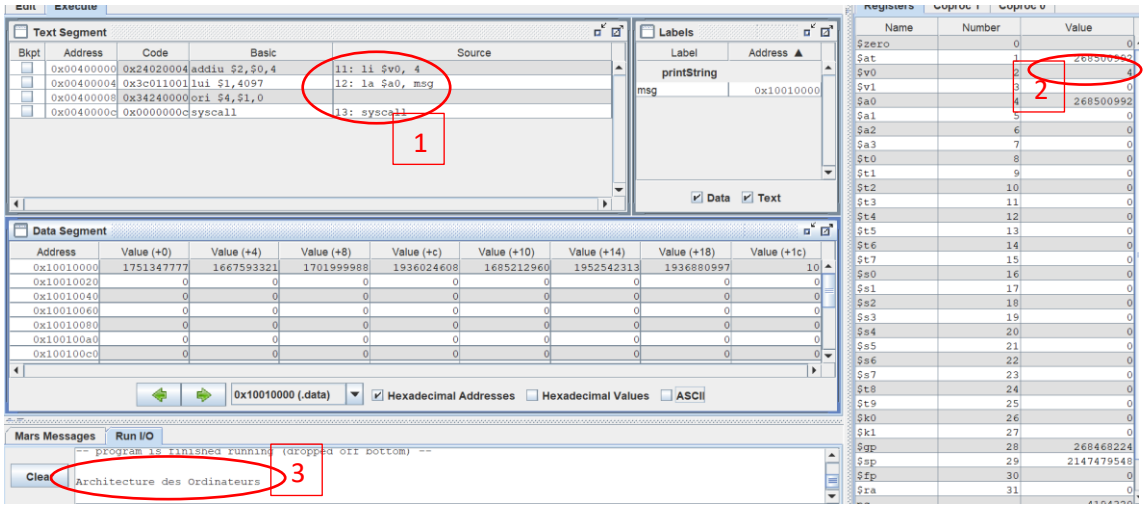


Figure 5.9: Printing a string of characters

5.2.3 Terminating a MIPS Program Properly

To terminate a MIPS program, you must load the system service number: 10 (which corresponds to **Exit**) into the $v0 register, then execute this system service using the **syscall** instruction.

Exercise 5.1: write a program that prompts the user for an integer, reads it, then squares that integer and prints the result with a message.

Solution:

Edit | Execute

carré.asm

```
1  # Computing the square of an integer
2  #-----------------------------------------------
3  .data
4  invite: .asciiz "Please enter an integer..."
5  result: .asciiz "The square of the input integer is: "
6  .text
7  li $v0, 4
8  la $a0, invite
9  syscall
10 li  $v0, 5      #reading an integer
11 syscall
12 mult $v0, $v0
13 # displaying the result
14 li $v0, 4
15 la $a0, result
16 syscall
17 li $v0, 1
18 mflo $a0
19 syscall
20 li $v0, 10      # terminating the program
21 syscall
```

The figure below presents a screenshot of an example execution for computing the square of an integer. The corresponding input and output are shown

Mars Messages | Run I/O

Clear

```
Please enter an integer...8
The square of the input integer is: 64
-- program is finished running --
```

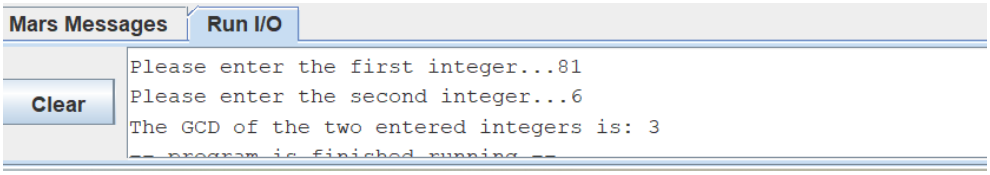
Exercise 5.2: Consider the pseudo-code below for a program that prompts the user to enter two integers (a and b), reads these two integers, and then calculates their greatest common divisor (GCD). At the end, this program prints the result (assuming that a > b).

```
Algorithm GCD ;
Input: Two integers a and b;
Output: a;
begin
write ('Please enter the first integer...');
read (a);
write ('Please enter the second integer...');
read (b);
while b ≠ 0
   t := b;
   b := a modulo b;
   a := t;
 endwhile
write ('The GCD of the two entered integers is: ', a);
end.
```

Solution: The listing below shows the source code proposed as a solution to the exercise. It computes the GCD of two given integers.

```
.data
msg1: .asciiz "Please enter the first integer..."
msg2: .asciiz "Please enter the second integer..."
msg_r : .asciiz "The GCD of the two entered integers is: "
.text
li $v0, 4
la $a0, msg1
syscall
li $v0, 5
syscall
move $s0, $v0
li $v0, 4
la $a0, msg2
syscall
li $v0, 5
syscall
move $s1, $v0
test: beq $s1, $0, end
      move $t0, $s1
      div $s0, $s1
      mfhi $s1
      move $s0, $t0
j test
end:
li $v0, 4
la $a0, msg_r
syscall
move $a0, $s0
li $v0, 1
syscall
li $v0, 10
syscall
```

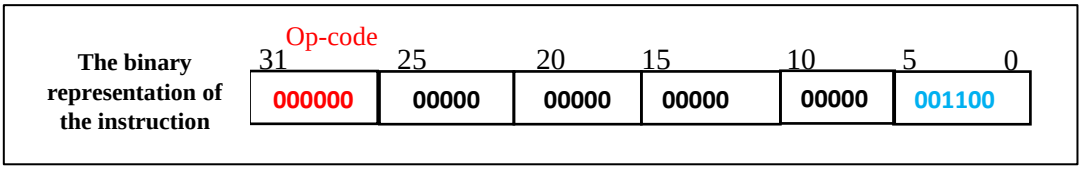
The figure below presents a screenshot of an example execution for computing the GCD of two integers (81 and 6).



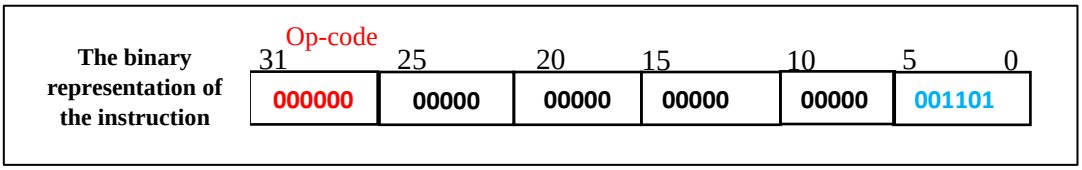
5.3 Special Instructions

The two instructions SYSCALL and BREAK that were presented in the section 5.1.3, are special instructions that allow the software to launch interrupts and are always of type R.

- **SYSCALL**: A system call exception occurs, immediately and unconditionally transferring control to the exception handler.
  Syntax: **syscall**



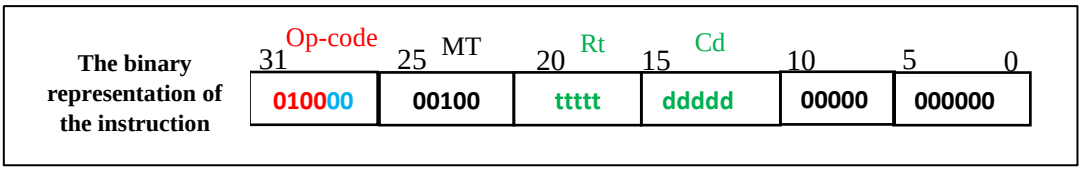
- **BREAK**: A breakpoint interrupt occurs, immediately and unconditionally transferring control to the exception handler.
  Syntax: **break**



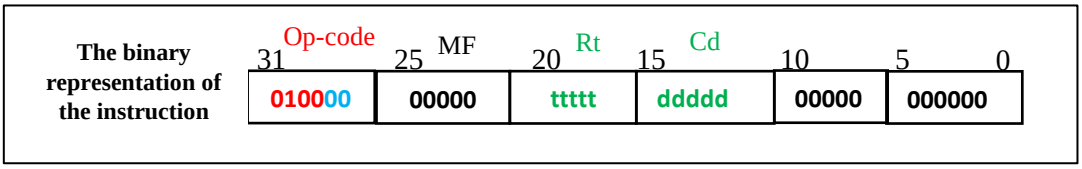
5.4 System Control Coprocessor (CP0) Instructions

Coprocessor 0 instructions perform operations on the System Control Coprocessor (CP0) registers to handle the memory management and exception handling features of the processor. Among others, we cite:

- **MTC0**: This instruction loads the specified coprocessor-0 register with a copy of the value in a general-purpose register. This instruction is used to handle exceptions.
  Syntax: **mtc0 $rt, $Cd**



- **MFC0**: This instruction loads the specified general register with a copy of the value found in a coprocessor 0 register.
  Syntax: **mfc0 $rt, $Cd**

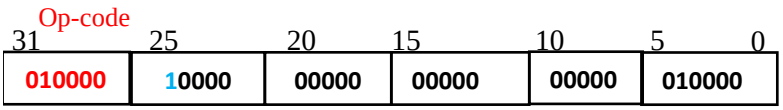


- **RFE**: This instruction restores the status register (SR) to the value it contained before the last exception occurred.

System Control Coprocessor (CP0) Instructions

Syntax: **rfe**

The binary representation of the instruction



Appendix: Presentation of the MARS simulator

"The MARS program is a combined assembly language editor, assembler, simulator, and debugger for the MIPS processor. It was developed by Pete Sanderson and Kenneth Vollmar of Missouri State University."

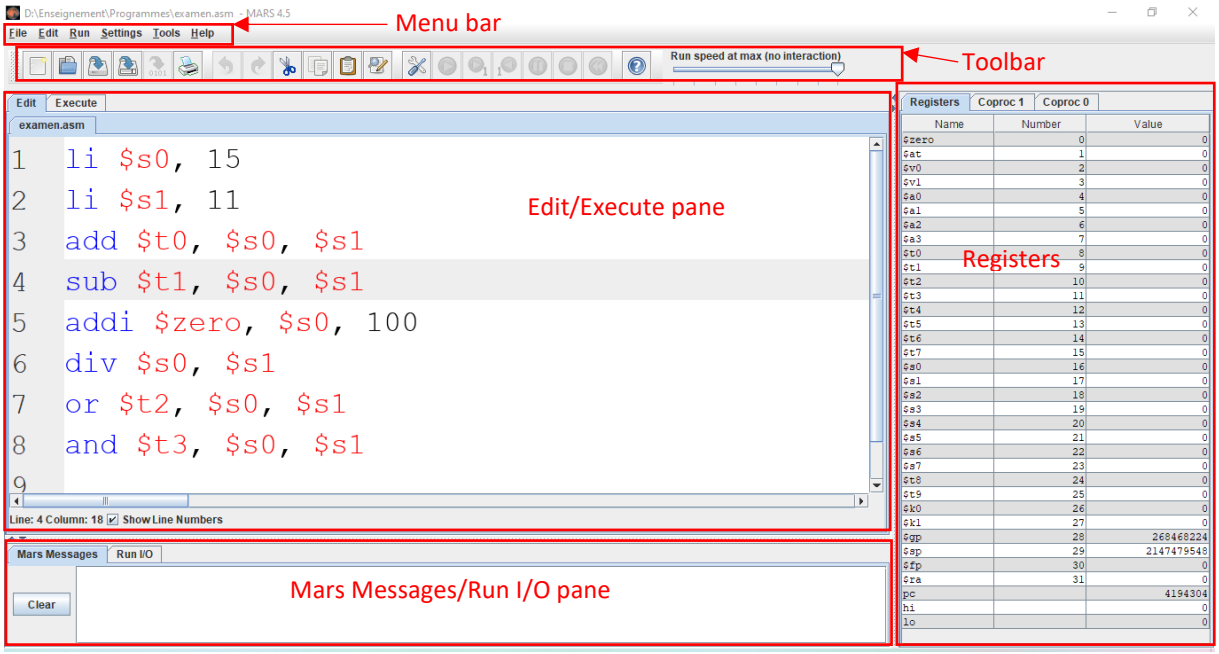
MARS was developed in Java and is distributed as an executable JAR file. Therefore, it requires the Java Runtime Environment (JRE) to run. At least version 1.5 of the J2SE Java Runtime Environment is required.

It can be used either (1) from a command line, e.g., `java –jar mars.jar`, (2) or via its integrated development environment (IDE), by double-clicking the mars.jar icon that represents this executable JAR file.

MARS User Interface

The MARS user interface can be divided into five main areas:

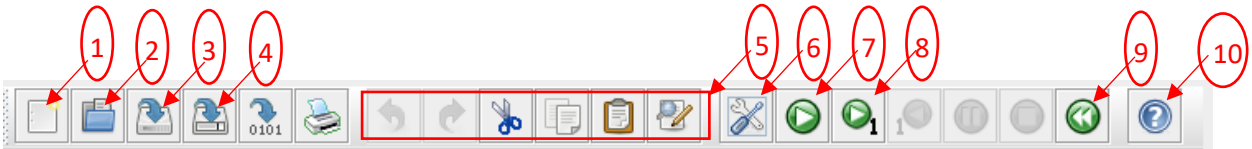
1. The menu bar contains menu items for performing various actions.
2. The toolbar contains buttons for the most commonly used menu items. They are grouped into four groups corresponding to the "File", "Edit", "Run", and "Help" menus.
3. The Registers pane is a tabbed pane. Unless you are writing floating-point systems or programs, you usually leave the "Registers" tab selected. This displays the entire registers of the simulated MIPS processor.
4. The Edit/Execute pane is a tabbed pane:
    - Clicking the "Edit" tab displays the MIPS assembly language editor.
    - Clicking the "Execute" tab displays assembly and runtime information.
5. The Mars Messages/Run I/O pane is a tabbed pane.
    - Clicking the "Mars Messages" tab displays the assembler and some runtime messages.
    - Clicking the "Execute I/O" tab displays the program's I/O and some runtime messages.



Basic operations

The toolbar contains buttons for frequently used operations during programming.

1. **New**: Creates a new MIPS assembly language file.
2. **Open**: Opens an existing MIPS assembly language file.
3. **Save**: Saves the program with the current name.
4. **Save as**: Saves the program with a new name.
5. A group of buttons for editing code, such as undo, redo, cut, copy, paste, find, and replace.
6. **Assemble**: Assembles a MIPS assembly language program.
7. **Go**: Executes a MIPS assembly language program.
8. **Step**: Executes one step at a time.
9. **Reset**: Resets the program and thus runs it again (by clicking Run) without reassembling.
10. **Help**: Displays MARS help.



Bibliography

1. A. S. Tanenbaum, *Structured Computer Organization*, 5th ed.: Pearson Prentice Hall, 2005.
2. S. Tanenbaum, *Architecture de l'ordinateur : du circuit logique au logiciel de base.,* Paris : InterEditions, 1987.
3. E. Lazard, *Architecture de l'ordinateur*, 2011.
4. A. Cazes, and J. Delacroix, *Architecture des machines et des systèmes informatiques*, 3e ed., 2008.
5. P. Amblard, *Architectures logicielles et matérielles : Cours, études de cas et exercices corrigés*, 2000.
6. G. Blanchet, and B. Dupouy, *Computer Architecture*: ISTE Ltd and John Wiley & Sons, Inc, 2013.
7. J. Y. Hsu, *Computer Architecture: Software Aspects, Coding, and Hardware*, 1st ed.: CRC Press, 2001.
8. W. Stallings, *Computer Organization and Architecture: Designing for Performance*, 8th ed.: Pearson Prentice Hall, 2010.
9. D. A. Patterson, and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 4th ed.: Morgan Kaufmann, 2012.
10. S. G. Shiva, *Computer Organization, Design, and Architecture*, 4th ed.: CRC Press, 2007.
11. L. Null, and J. Lobur, *The essentials of computer organization and architecture*: Jones and Bartlett, 2003.
12. M. Abd-El-Barr, and H. El-Rewini, *Fundamentals of Computer Organization and Architecture*: John Wiley & Sons, Inc, 2005.
13. D. E. Comer, *Essentials of Computer Architecture*, 1st ed.: Pearson, 2005
14. M. M. Mano, *Computer System Architecture*, 3rd ed.: Pearson
15. H. G. Cragon, *Computer Architecture and Implementation*. Cambridge: Cambridge University Press, 2000.
16. *IDT R30xx Family Software Reference Manual*: Integrated Device Technology (IDT), 1994.
17. R. L. Britton, *MIPS Assembly Language Programming*: Pearson, 2003.
18. E. Jorgensen, *MIPS Assembly Language Programming using QtSpim*: Ed Jorgensen, 2017.
19. G. Kane, and J. Heinrich, *MIPS RISC Architecture*, 2nd ed.: Prentice Hall, 1991.
20. P. Bazargan, F. Dromard, and A. Greiner, *Processeur MIPS R3000. Architecture Interne Microprogrammée*, UNIVERSITÉ PIERRE ET MARIE CURIE, 2002.
21. *Processeur MIPS R3000. Architecture externe*, UNIVERSITE PIERRE ET MARIE CURIE, 2003.
22. *Processeur MIPS R3000. Langage d'assemblage*, UNIVERSITÉ PIERRE ET MARIE CURIE, 2001.
23. D. Sweetman, *See MIPS Run*, 2nd ed., 2006.
24. I. BEKKOUCHE, "ARCHITECTURE DES ORDINATEURS," USTO-MB, ed., 2020-2021.
25. G. Blin, Introduction à l'Architecture des ordinateurs, lecture notes, Université Paris-Est Marne-la-Vallée. [Online]. Available: http://igm.univ-mlv.fr/ens/Licence/L3/2008-2009/ArchiOrdi/cours/BlinGuillaume-ArchiOrdi-Intro.pdf
26. https://courses.missouristate.edu/kenvollmar/mars/Help/MarsHelpIntro.html
27. https://www.d.umn.edu/~gshute/mips/Mars/Mars.html
28. https://digitalcommons.otterbein.edu/cgi/viewcontent.cgi?article=1004&context=math_fac