Democratic and Popular Republic of Algeria
Ministry of Higher Education and Scientific Research
Abderrahmane Mira University of Bejaia

Faculty of Exact Sciences
Department of Computer Science

Lecture Notes

Computer architecture

Created by:

Dr Mohammed KHAMMARI

Year 2024

Contents

List of Figures

List of Acronyms

- **ALU :** Arithmetic Logic Unit.

- **CU :** Control Unit.

- **I/O :** Input/Output.

- **CPU :** Central Processing Unit.

- **PC :** Program Counter.

- **RAM :** Random-Access Memory.

- **DRAM :** Dynamic Random-Access Memory.

- **SRAM :** Static Random-Access Memory.

- **SSD :** Solid-State Drives.

- **MRAM :** Magneto-resistive Random-Access Memory.

- **FERAM :** Ferroelectric Random-Access Memory.

Preface

Welcome to the Fascinating World of Computer Architecture! Welcome to Computer Architecture, an exciting journey into the inner workings of the machines that power our digital world. This course is designed for second-year Licence students, eager to delve deeper into the intricate dance between hardware and software that brings computers to life. Get ready to embark on a five–chapter adventure, each unveiling a crucial layer of the computational onion:

- **Chapter 1**: Introduction to Computer Architecture: Laying the foundation, we'll explore the fundamental concepts that define computer architecture, its history, and how it shapes the performance and capabilities of the devices we rely on daily.
- **Chapter 2**: Main Components of a Computer: Dive into the heart of the machine, examining the key players like the CPU, memory, I/O devices, and their intricate interplay. You'll gain a tangible understanding of how these components orchestrate the digital symphony.
- **Chapter 3**: Basics of Computer Instructions: Understand the language of machines! We'll decode the secret messages, called instructions, that computers execute to perform tasks. You'll learn how these instructions are structured, fetched, and executed, orchestrating the computational magic.
- **Chapter 4**: MIPS R3000: Put theory into practice! We'll delve into the specifics of a real-world processor, the MIPS R3000, dissecting its architecture and instruction set. This hands-on approach will solidify your understanding and equip you with valuable practical skills.
- **Chapter 5**: Special Instructions : In this chapter, we'll explore special instructions that go beyond the basic operations covered in Chapter 3. These specialized

instructions provide essential functionalities for tasks like handling interrupts, managing memory access, and controlling system operations. Understanding these instructions will enhance your grasp of the capabilities of a modern processor.

Beyond the chapters, this course is designed to be an interactive experience. Prepare to engage in:

- Weekly Lab Sessions: Get your hands dirty with real-world computer architecture. You'll work on practical exercises and simulations, solidifying your theoretical understanding and gaining valuable technical skills.
- Weekly tutorial: Dive deeper into specific topics that pique your interest. These guided explorations will allow you to personalize your learning journey and delve into cutting-edge areas of computer architecture.
- Weekly lecture: No computer works in isolation! We'll foster a collaborative learning environment where you can share your insights, ask questions, and learn from your peers.

This course is your gateway to understanding the invisible forces that drive the digital world. By the end of your journey, you'll possess a deep appreciation for the intricate dance between hardware and software, the elegance of computer architecture, and the power you hold to shape the future of computing.

So, buckle up, get ready to explore, and let's delve into the captivating world of Computer Architecture!

Best regards,

Dr. Mohammed Khammari
Instructor, Computer Architecture

Chapter 1

Introduction to Computer Architecture

1.1 Introduction

A computer is a machine that can be programmed to carry out a set of instructions. It is a general-purpose device that can be used for a variety of tasks, including calculations, data processing, and communication.

1.2 History of the computer

The history of the computer can be traced back to the 19th century, when early mechanical computers were developed. These computers were very limited in their capabilities, but they laid the foundation for the development of modern computers.

The first electronic computer, the ENIAC, was built in the United States during World War II. The ENIAC was a huge machine that weighed 30 tons and took up an entire room. It was used to calculate ballistics tables for the US Army.

In the 1950s, the first commercial computers were developed. These computers were much smaller and more affordable than the ENIAC, and they quickly became popular in business and government.

In the 1960s, the first minicomputers were developed. These computers were even smaller and more affordable than the first commercial computers, and they made computing accessible to a wider range of people.

In the 1970s, the first personal computers were developed. These computers were designed for individual use, and they revolutionized the way people interact with computers.

In the 1980s, the personal computer market exploded. Computers became more affordable and powerful, and they became an essential part of everyday life.

In the 1990s, the Internet was developed. The Internet made it possible for computers to connect to each other and share information. This led to the development of new applications and services, such as e-mail, the World Wide Web, and social media.

1.3 Von Neumann and Harvard architectures

There are two main types of computer architectures: the Von Neumann architecture and the Harvard architecture. The Von Neumann architecture (Figure 1.1) is the most common type of computer architecture. In a Von Neumann architecture, the program instructions and data are stored in the same memory. This makes it possible for the computer to fetch the next instruction and the next data item from the same memory location. The Harvard architecture (Figure 1.2) is a variation of the Von Neumann architecture. In a Harvard architecture, the program instructions and data are stored in separate memories. This makes it possible for the computer to fetch the next instruction and the next data item from different memory locations. Examples of Von Neumann and Harvard architecture computers

- **Von Neumann architecture computers:** The ENIAC, the IBM 360, the Apple II, the IBM PC, the Macintosh
- **Harvard architecture computers:** The Mark I, the UNIVAC I, the HP 9100A, the PDP-8, the VAX

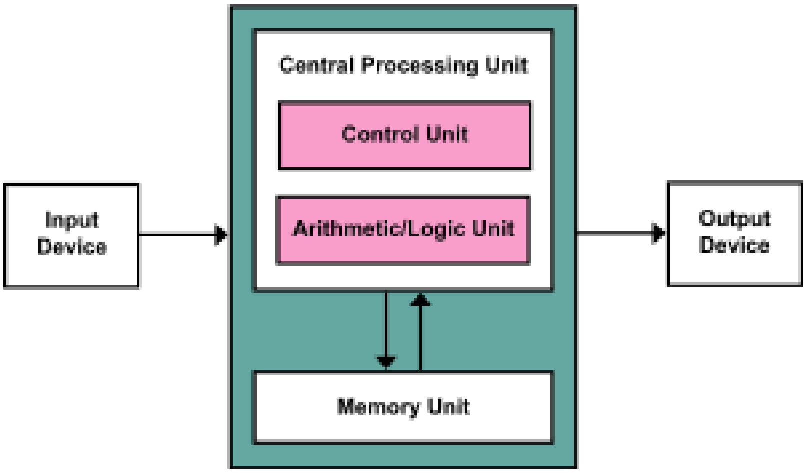


Figure 1.1: Von Neumann architecture [4]

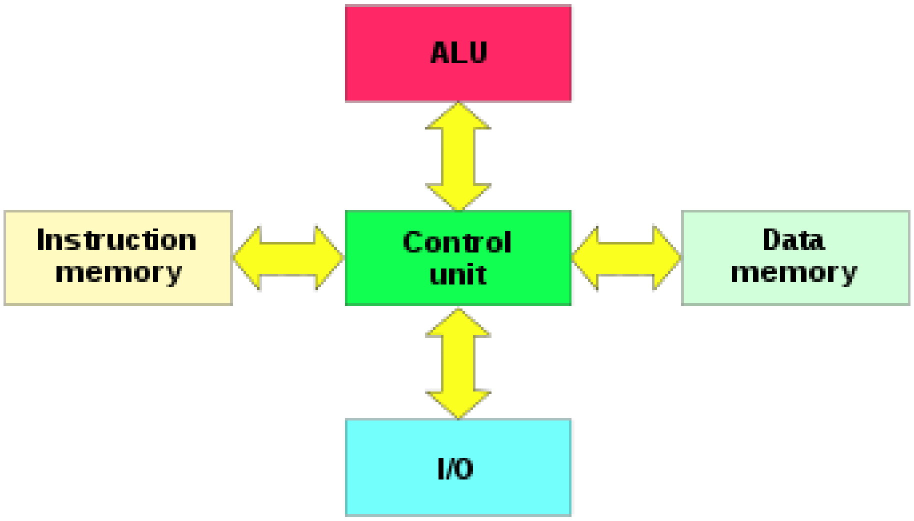


Figure 1.2: Harvard architecture [3]

1.4 Conclusion

Computers have come a long way since their early beginnings. They are now ubiquitous in our lives, and they continue to evolve at a rapid pace.

Chapter 2

Main Components of a Computer

2.1 Introduction

The Von Neumann architecture is a computer architecture that was first proposed by John von Neumann in the 1940s. It is a stored-program computer architecture, which means that both the instructions and the data are stored in the same memory. This is in contrast to earlier computer architectures, which used separate memories for instructions and data.

2.2 The global architecture of Von Neumann

The Von Neumann architecture is based on the following four components:

- **Arithmetic logic unit (ALU):** The ALU performs arithmetic and logical operations on data.
- **Control unit (CU):** The CU controls the flow of instructions through the computer. It fetches instructions from memory, decodes them, and sends them to the ALU to be executed.
- **Memory:** Memory stores both instructions and data.
- **Input/output (I/O):** I/O devices allow the computer to communicate with the outside world.

The Von Neumann architecture works as follows:

1) The CU fetches an instruction from memory.

2) The CU decodes the instruction and sends it to the ALU.

3) The ALU executes the instruction, which may involve performing arithmetic or logical operations on data.

4) The results of the instruction are stored in memory or sent to an I/O device.

5) The CU fetches the next instruction from memory and repeats the process.

2.2.1 Central processing unit (CPU)

The CPU is the brain of the computer. It is responsible for executing instructions, which are the commands that tell the computer what to do. The CPU is a complex piece of hardware that is made up of many different components.

The main components of the CPU are:

2.2.1.1 Control unit (CU)

The CU (Figure 2.1) is the brain of the Central Processing Unit (CPU). It is responsible for directing the flow of instructions through the CPU and ensuring that they are executed correctly. The CU does this by performing the following tasks:

- **Fetching instructions from memory:** The CU fetches instructions from memory one at a time.

- **Decoding instructions:** The CU decodes the instructions to determine what action the CPU needs to take.

- **Sending signals to the ALU:** The CU sends signals to the arithmetic logic unit (ALU) to tell it what operation to perform on the data.

- **Controlling the flow of data:** The CU controls the flow of data between the ALU, memory, and registers.

- **Controlling I/O:** The CU controls the input/output (I/O) devices by sending signals to them to tell them when to read or write data.

The CU is a complex piece of hardware that is made up of many different components. However, the basic principles of its operation are relatively simple.

- **Here is a more detailed explanation of how the CU works:**

1. The CU fetches an instruction from memory.

2. The CU decodes the instruction to determine the following:

  o What operation needs to be performed (e.g., addition, subtraction, multiplication, division, etc.)

  o What data needs to be operated on (e.g., the contents of a register or a memory location)

  o Where to store the result of the operation (e.g., a register or a memory location)

3. The CU sends signals to the ALU to tell it what operation to perform on the data.

4. The ALU performs the operation and stores the result in a register or memory location.

5. The CU repeats steps 1-4 until it encounters a halt instruction.

The CU is a critical component of the CPU, and it plays a vital role in the operation of the computer. Without the CU, the CPU would not be able to execute instructions and the computer would not be able to function.

- **Here are some examples of instructions that the CU might execute:**

o Load a value from memory into a register.

o Add two values in registers and store the result in a register.

o Compare two values in registers and set a flag based on the result.

o Jump to a different location in the instruction stream.

o Call a subroutine.

The CU is responsible for ensuring that all of these instructions are executed correctly. It does this by keeping track of the current state of the CPU and by sending the appropriate signals to the ALU and other components.

The CU is a complex piece of hardware, but it is essential for the operation of the computer. By understanding how the CU works, you can better understand how computers work in general.

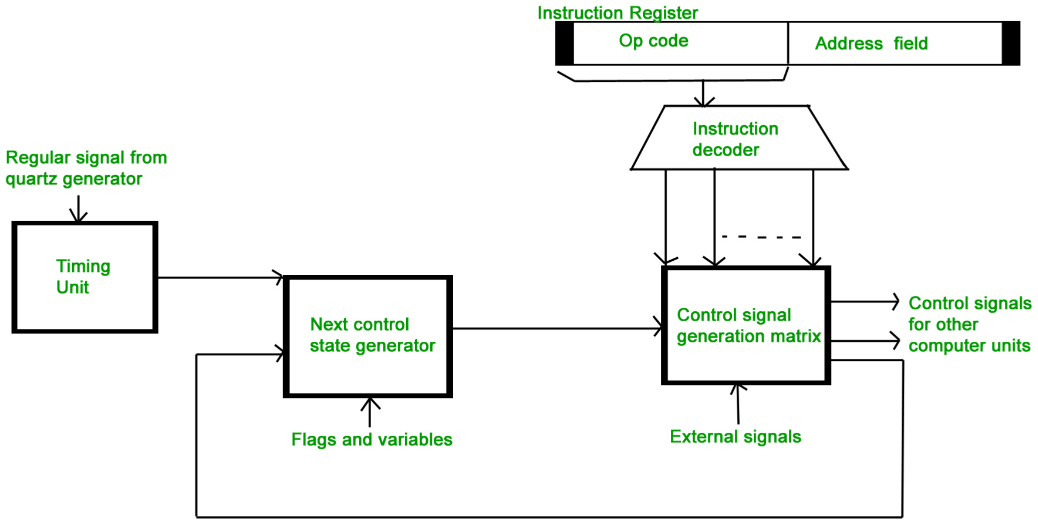

Instruction Register
Op code
Address field
Instruction decoder
Regular signal from quartz generator
Timing Unit
Next control state generator
Control signal generation matrix
Control signals for other computer units
Flags and variables
External signals

Figure 2.1: Control Unit (CU) [13]

2.2.1.2 Arithmetic logic unit (ALU):

The ALU (Figure 2.2) is the part of the CPU that performs arithmetic and logical operations on data. It is responsible for performing all of the calculations that the computer needs to do, such as adding, subtracting, multiplying, and dividing numbers. The ALU is also responsible for performing logical operations, such as AND, OR, and NOT. The ALU is a very important part of the CPU, and its performance can have a significant impact on the overall performance of the computer. For this reason, ALU design is a very active area of research.

- **How the ALU works:**

The ALU works by fetching two operands from memory or registers, performing the desired operation on them, and storing the result in memory or a register. The operands can be numbers, logical values, or even characters. The ALU can perform a wide variety of operations, including:

- Arithmetic operations: addition, subtraction, multiplication, division, modulo, and square root.
- Logical operations: AND, OR, NOT, XOR, and shifts.
- Comparison operations: equal to, not equal to, greater than, less than, greater than or equal to, and less than or equal to.

The ALU is controlled by the control unit (CU). The CU tells the ALU what operation to perform on the operands and where to store the result.

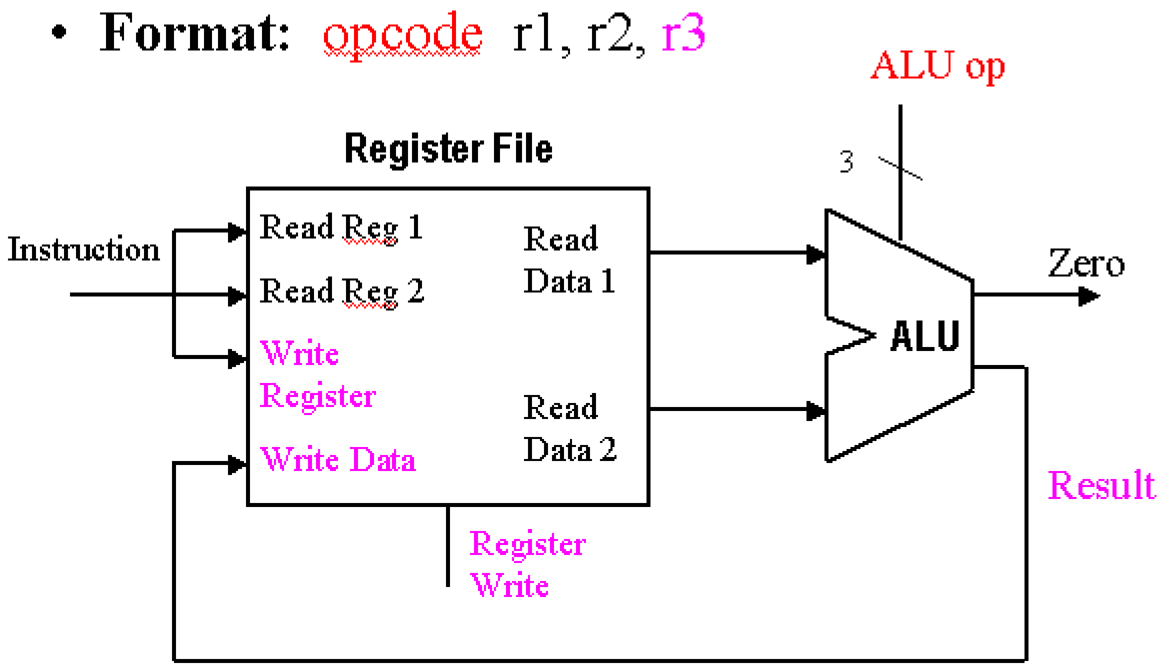


Figure 2.2: Arithmetic Logic Unit (ALU) [13]

2.2.1.3 Registers:

Registers are small, fast storage units that are part of a computer's central processing unit (CPU). They are used to store data and instructions that are being used by the CPU. Registers are located on the CPU chip, which is the part of the computer that does the actual computing. They are much faster than main memory, which is the larger, slower memory that stores all of the computer's data and programs.

- **Registers are used for a variety of purposes, including:**

o Storing data that is being used in an arithmetic or logical operation

o Storing the address of the next instruction to be executed

o Storing the results of an arithmetic or logical operation

o Storing flags, which are bits that indicate the status of the CPU

- **Types of registers:**

There are two main types of registers:

General-purpose registers: are the most common type of register. They can be used to store data of any type, including integers, floating-point numbers, and addresses.

Special-purpose registers: are designed for a specific purpose. Some common types of special-purpose registers include: Accumulator registers: are used to store the results of arithmetic operations. Program counter (PC): registers store the address of the next instruction to be executed. Flags registers: store bits that indicate the status of the CPU, such as whether a carry or borrow occurred during an arithmetic operation.

2.2.1.4 Memory:

Main memory, also known as random-access memory (RAM), is the memory that is directly accessible to the CPU. It is the fastest and most expensive type of memory in a computer. Main memory is used to store: the operating system, the currently running programs, data that is being processed by the CPU

The two main types of main memory are dynamic random-access memory (DRAM) and static random-access memory (SRAM).

- **Dynamic Random-Access Memory (DRAM):** is the most common type of main memory. It is less expensive than Static Random-Access Memory (SRAM), but it is also slower. DRAM stores data in capacitors, which lose their charge over time. This requires the capacitors to be refreshed periodically. DRAM is divided into two main types: Synchronous DRAM (SDRAM) and Double Data Rate (DDR) DRAM. SDRAM is the older type of DRAM, but it is still used in some older computers. DDR DRAM is faster than SDRAM and is used in most modern computers.
- **Static Random-Access Memory (SRAM):** is faster than DRAM, but it is also more expensive. SRAM stores data in flip-flops, which do not lose their state over time. This means that SRAM does not need to be refreshed. SRAM is used in applications where speed is more important than cost, such as in the cache memory of a computer's CPU.

Other types of main memory:

In addition to DRAM and SRAM, there are a number of other types of main memory that are used in specialized applications. These include:

o **Flash memory:** is a type of non-volatile memory, which means that it retains its data even when the power is turned off. Flash memory is used in a variety of applications, such as USB drives, solid-state drives (SSDs), and memory cards.

o **Magneto-resistive random-access memory (MRAM):** is a type of non-volatile memory that uses magnetic fields to store data. MRAM is faster than DRAM and SRAM, but it is also more expensive. MRAM is still under development, but it has the potential to replace DRAM and SRAM in the future.

o **Ferroelectric random-access memory (FERAM):**): is a type of non-volatile memory that uses ferroelectric materials to store data. FERAM is faster than

DRAM and SRAM, but it is also more expensive. FERAM is also still under development, but it has the potential to replace DRAM and SRAM in the future.

Cache memory

Cache memory is a small amount of very fast memory that is located between the CPU and main memory. It stores copies of the most frequently accessed data and instructions from main memory. This allows the CPU to access this data and instructions much faster than if it had to go to main memory every time. Cache memory is typically implemented as a hierarchy, with multiple levels of cache memory. Each level of cache is smaller and faster than the previous level.

The smallest and fastest level of cache is called the L1 cache. It is located right next to the CPU and stores copies of the most frequently accessed data and instructions. The next level of cache is called the L2 cache. It is larger and slower than the L1 cache, but it still much faster than main memory. The L2 cache stores copies of data and instructions that are not as frequently accessed as the data and instructions in the L1 cache.

Some CPUs also have a third level of cache, called the L3 cache. The L3 cache is the largest and slowest level of cache, but it is still much faster than main memory. The L3 cache stores copies of data and instructions that are even less frequently accessed than the data and instructions in the L2 cache.

• Reading and writing to main memory

The CPU can read and write to main memory using a process called memory access. Memory access is initiated by the CPU when it needs to read or write data from or to main memory. The CPU sends a memory access request to the memory controller, which is a small chip that controls access to main memory. The memory controller then sends the memory access request to the main memory. The main memory responds to the memory access request by returning the requested data to the CPU or by writing the requested data to main memory. The memory controller then returns the response to the CPU.

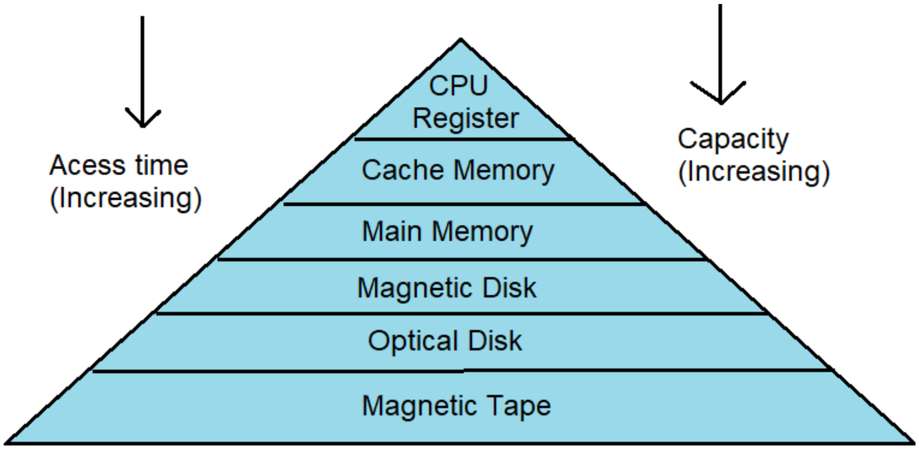


Figure 2.3: Memory Hierarchy [12]

Little endian and **big endian** are two ways of storing data in memory. Little endian stores the least significant byte (LSB) first, while big endian stores the most significant byte (MSB) first. To read from main memory with little endian, the CPU will read the bytes from memory in the order that they are stored, starting with the LSB. To write to main memory with little endian, the CPU will write the bytes to memory in the order that they are given, starting with the LSB. To read from main memory with big endian, the CPU will read the bytes from memory in the reverse order that they are stored, starting with the MSB. To write to main memory with big endian, the CPU will write the bytes to memory in the reverse order that they are given, starting with the MSB.

Here is an example of how little endian and big endian work with 0x 11223344. To read or write the value to memory, the CPU would write the bytes in the following order:
Little endian: 0x11, 0x22, 0x33, 0x44
Big endian: 0x44, 0x33, 0x22, 0x11

Why is little endian more common?

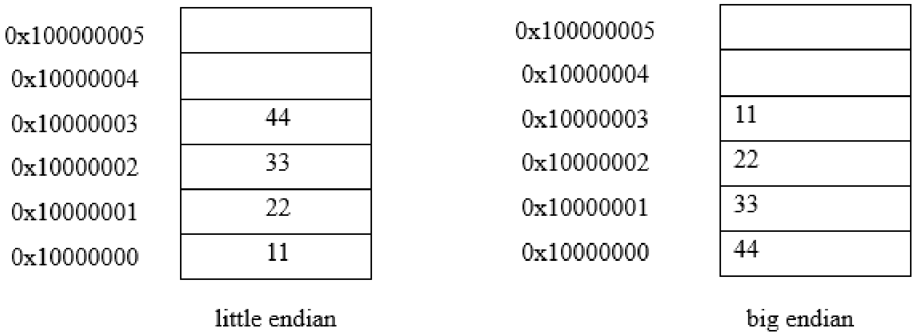


Figure 2.4: Little endian and Big endian

Little endian is more common than big endian because it is more efficient for most processors. Most processors are designed to read and write data in word-sized chunks. A word is typically 32 or 64 bits wide. When reading a word from memory, the processor can read the bytes in the order that they are stored and then assemble them into a word. When writing a word to memory, the processor can disassemble the word into bytes and then write the bytes to memory in the order that they need to be stored. Little endian is also more efficient for networking because most network protocols store data in little endian format. This means that when a computer is sending data over a network, it does not need to convert the data from big endian to little endian before sending it.

2.2.1.5 Input and Output (I/O)

Input and output (I/O) are the processes of transferring data between a computer and the outside world. Input is the process of bringing data into the computer, while output is the process of sending data out of the computer.

- **I/O devices**

I/O devices are the hardware components that allow the computer to interact with the outside world. There are many different types of I/O devices, including:

o Keyboards and mice: These devices are used to enter data into the computer.

o Monitors: These devices are used to display data on the screen.

o Printers: These devices are used to print data on paper.

o Storage devices: These devices are used to store data on a permanent basis.

o Scanners: These devices are used to convert images and text into digital form.

o Network adapters: These devices allow the computer to connect to a network.

2.2.1.6 The bus concept

In computer architecture, a bus is a communication pathway that allows data to be transferred between different components of a computer system. There are many different types of buses, each with its own purpose and characteristics.

- **Data buses:** Data buses are used to transfer data between the CPU and other components of the computer, such as memory, I/O devices, and storage devices. Data buses are typically parallel, meaning that they transfer data one bit at a time on multiple wires. The width of a data bus is measured in bits, and it determines how much data can be transferred at once. For example, a 32-bit data bus can transfer 32 bits of data at once, which is equivalent to 4 bytes.

- **Address buses:** Address buses are used to identify the memory location where data is stored. Address buses are typically parallel, and they use a different wire for each bit of the address. The width of an address bus is measured in bits, and it determines how many memory locations can be addressed. For example, a 32-bit address bus can address 2^{32} memory locations, which is equivalent to 4 GB of memory.

- **Control buses:** Control buses are used to send control signals between the CPU and other components of the computer. Control signals are used to initiate data transfers, control the flow of data, and provide feedback from the devices. Control buses are typically serial, meaning that they transfer one signal at a time.

- **Other types of buses:** In addition to data, address, and control buses, there are many other types of buses used in computer systems. Some common examples include:

  o Interrupt buses: These buses are used to send interrupt signals from I/O devices to the CPU.

- **o** Timing buses: These buses are used to synchronize the operation of different components of the computer system.
- **o** Power buses: These buses are used to supply power to the components of the computer system. etc

The main components of a computer are shown in Figure 2.5

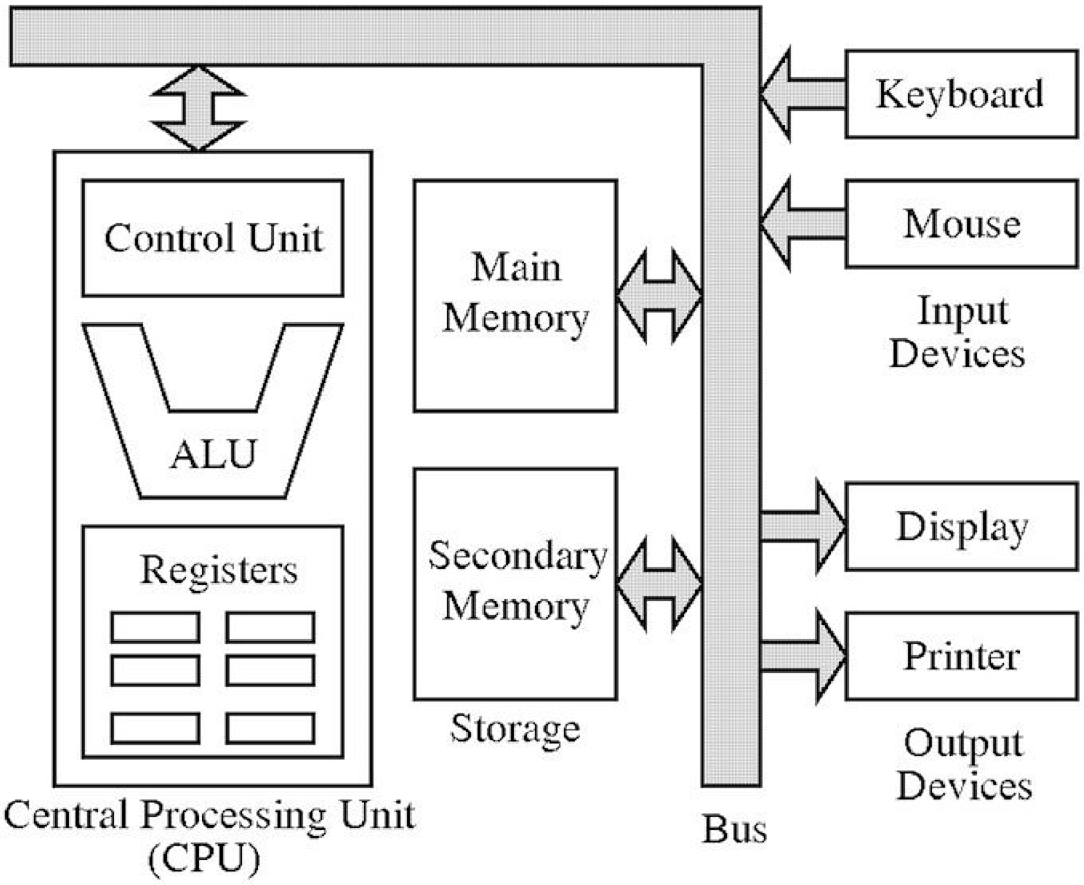


Figure 2.5: Main components of a computer [6]

2.3 Conclusion

The main components of a computer are the CPU, memory, I/O devices, and buses. These components work together to allow the computer to perform its tasks.

The CPU is the central processing unit, and it is responsible for executing the instructions that make up a computer program. Memory is used to store data and instructions, and I/O devices allow the computer to interact with the outside world. Buses are used to connect the different components of the computer and allow them to communicate with each other.

By understanding the main components of a computer, you can gain a better understanding of how computers work. This knowledge can be useful for a variety of purposes, such as troubleshooting computer problems, upgrading computer components, and developing computer software.

Chapter 3

Basics of Computer Instructions

3.1 Introduction

An instruction is a command that tells a computer what to do. Instructions are the basic building blocks of computer programs. They are used to perform arithmetic operations, logical operations, control the flow of execution, and interact with input and output devices. Instructions are typically encoded in machine language, which is a low-level language that is directly understood by the computer's central processing unit (CPU). Machine language instructions are typically very short and concise, and they are typically represented as a sequence of binary digits.

In this chapter, we will discuss the different types of instructions, the different formats of instructions, and how instructions are executed

3.2 Languages of Programming

There are three main types of programming languages: high-level languages, assembly languages, and machine languages. High-level languages are designed to be easy for humans to read and write. They are typically translated into machine language by a compiler or interpreter.

Assembly languages are a type of low-level language that is designed to be more efficient than high-level languages. They are typically translated into machine language by an assembler. Machine languages are the lowest-level type of programming language. They are directly understood by the computer's CPU.

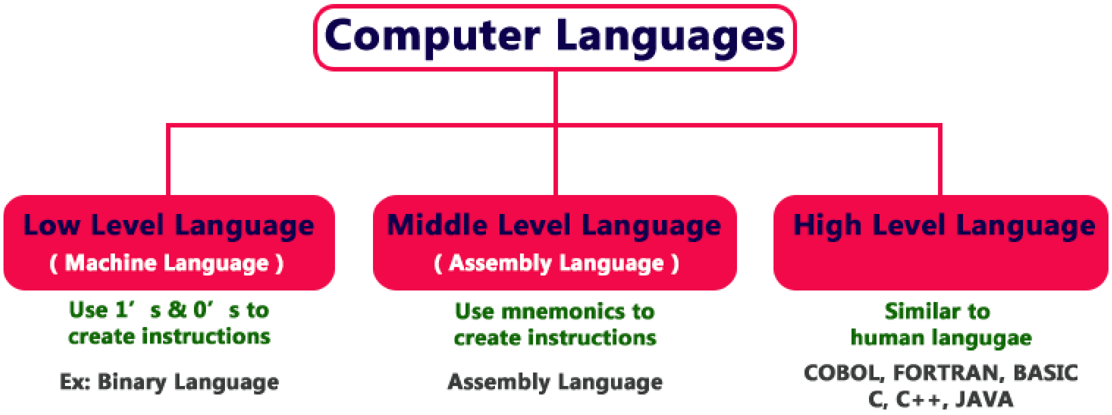


Figure 3.1: Computer Languages Classification (a). [11]

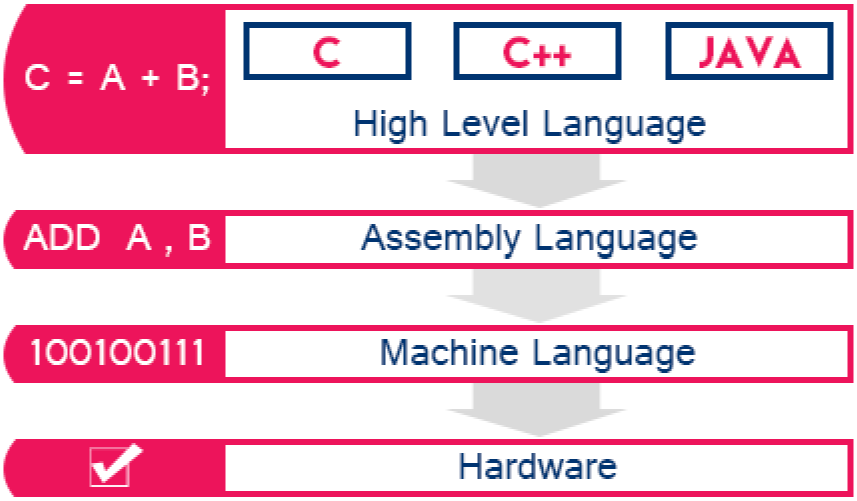


Figure 3.2: Computer Languages Classification (b). [11]

3.3 Modes of Addressing

Addressing is how an instruction in a computer program specifies the location of data in memory. Different addressing modes allow programmers to control how data is accessed.

- **Immediate:** In immediate addressing, the operand value is specified directly within the instruction. For example, the instruction ADD R1, 10 adds the immediate value 10 to the content of register R1.
- **Direct:** In direct addressing, the operand value is specified by its address in memory. For example, the instruction LOAD R1, [0x100] loads the value stored at memory address 0x100 into register R1.
- **Indirect:** In indirect addressing, the operand value is specified by the address of another address. This address can be stored in a register or at another memory location. For example, the instruction LOAD R1, [R2] loads the value stored at the address of the address contained in register R2 into register R1.
- **Register:** In register addressing, the operand value is specified by the number of the register containing it. For example, the instruction ADD R1, R2 adds the content of @[2] to the content of register R1.
- **Indexed:** In indexed addressing, the operand value is specified by the address contained in a register Indexe, plus a displacement. For example, the instruction LOAD R1, [10] loads the value stored at the address contained in register Indexe, plus 10, into register R1.

Examples: Here are some examples of using different addressing modes:

addresses	Central Memory
10	100
20	50
30	500
40	700
50	80
60	30

Immediate:
ADD R1, 10 adds 10 to the content of register R1.
SUB R2, 5 subtracts 5 from the content of register R2.

Direct:
LOAD R1, [0x10] loads the value stored at memory address 0x10 into register R1 (R1 = 100).
Indirect:
LOAD R1, [0x20] loads the value stored at the address of the address into register R1 (R1 = 80).
Register:
ADD R1, R30 adds the content of @[30] to the content of register R1 (R1 = R1 + 500).
SUB R1, R20 subtracts the content of @[20] from the content of register R1 (R1 = R1 - 50).
Indexed:
LOAD R1, [10] loads the value stored at the address contained in register Indexe, plus 10, into register R1.(Indexe = 30, R1 = @[30+10] = 700)

3.4 Types of machines

- **Processors with general-purpose registers:** Processors with general-purpose registers can have instructions with the following number of address fields:
  - **3 address fields:** destination, source1, source2 (e.g., add R1, R2, R3 which calculates R1 <- R2 + R3)
  - **2 address fields:** the register of one of the operands receives the result (e.g., add R1, R2 which calculates R1 <- R1 + R2)
- **Processors with an Accumulator:** Processors with an accumulator have two specific instructions for accessing the accumulator: load X to load the accumulator with a data X, and store X to write the content of the accumulator to memory or another register.
  - **1 address field:** source2 (e.g., add R1 which calculates Acc <- Acc + R1)
- **Processors with a Stack:** A stack is a data structure in main memory in which data is placed "on top of each other": the last data entering the stack will be the

first to leave. A special register SP contains the address of the top of the stack. Two operations are defined on stack structures: push X: push a data X, that is, place it on top of the stack and advance the SP register to the next memory cell. pop X: pop to a memory cell or a register X, that is, place in X the data at the top of the stack designated by SP and advance the SP register to the previous memory cell. In this case, all operations take their operands from the top of the stack, and store the result at the top of the stack. In addition, the operands are automatically popped off.

- **0 address field:** (e.g., add which performs the calculation [SP-2] <- [SP-2] + [SP-1]; SP <- SP - 1 (in the case of a stack that grows towards high addresses))

Example :

Here is an example of how these instructions can be used:

```
// Processor with general-purpose registers
add R1, R2, R3 // R1 = R2 + R3
add R1, R2 // R1 = R1 + R2

// Processor with an accumulator
load 10 // acc = 10
add R1 // Acc = Acc + R1

// Processor with a stack
push R1 // Push R1 onto the stack
push R2 // Push R2 onto the stack
add // pop R1, pop R2, top of the stack = R1+R2
```

3.5 Common Machine Instructions

Assembler is a low-level language that gives you direct control over the processor. It's powerful, but complex and requires careful attention to detail. Learn assembler to unlock the full potential of your computer.

- **Basic Instructions** :
  - **Data transfer:** Load and store instructions move data between registers and memory.
  - **Arithmetic:** : Addition (add), subtraction (sub), multiplication (mult), and division (div) operations on various data types like integers and floating-point numbers.
  - **Logical:** AND, OR, NOT, and other logical operations on bits and registers.
  - **Comparison:** Compare two values and set flags based on the result (e.g., equal, greater than).
- **Control Flow** :
  - **Conditional jumps:** Branch to a different instruction based on the value of flags or a register.
  - **Loops:** Loop instructions allow for repeated execution of a block of code.
  - **Subroutines:** Call and return instructions for modular program execution.
- **Advanced Instructions** :
  - **Shift and rotate:** Shift data left or right by a specified number of bits.
  - **Bit manipulation:** Set, clear, or extract individual bits within a register.
  - **Floating-point arithmetic:** Specialized instructions for efficient floating-point computations.
  - **Memory management:** Instructions for handling virtual memory and memory protection.
- **Additional points to consider** :
  - Different architectures have different register sizes and types.
  - Some instructions operate on immediate values instead of register values.
  - Instructions can have varying addressing modes for specifying operand locations.
  - Optimization techniques can impact instruction selection and code efficiency.

This is just a basic framework. Specific architectures will have their own unique instructions and features. Remember, assembler programming requires careful attention to detail and understanding of the underlying hardware. Get ready to master the MIPS R300 instruction set in the next chapter! We'll use various examples and exercises to put your understanding to the test.

3.6 Instruction Execution cycle

The instruction execution cycle is the sequence of steps that a processor takes to execute an instruction. It is a fundamental concept in computer architecture and is essential for understanding how assembly programs work.

The instruction execution cycle can be divided into four main steps:

1. **Fetch:** The fetch step is the first step in the instruction execution cycle. In this step, the processor reads the instruction from memory. The address of the instruction is typically stored in the program counter (PC) register. The instruction is typically stored in memory as a sequence of bits. The processor decodes these bits to determine what the instruction does.

2. **Decode:** The decode step is the second step in the instruction execution cycle. In this step, the processor decodes the instruction to determine what it does. The instruction format varies depending on the processor architecture. However, most instructions have a common format that includes the following fields:

   - **Opcode:** This field specifies the operation that the instruction performs.
   - **Operands:** These fields specify the data that the instruction operates on.
   - **Addressing modes:** These fields specify how the operands are located in memory.

   The processor uses the opcode field to determine what operation to perform. The operands field specifies the data that the instruction operates on. The addressing modes field specifies how the operands are located in memory.

3. **Execute:** The execute step is the third step in the instruction execution cycle. In this step, the processor executes the instruction. The specific steps involved in the execute step vary depending on the instruction. However, some common instruction types include:

   - **Data transfer:** These instructions transfer data between registers or between memory and registers.
   - **Arithmetic:** These instructions perform arithmetic operations on data.
   - **Logical:** These instructions perform logical operations on data.
   - **Control flow:** These instructions control the flow of execution.

4. **Writeback:** The writeback step is the fourth and final step in the instruction execution cycle. In this step, the processor writes back the result of the instruction to memory or a register. The specific steps involved in the writeback step vary depending on the instruction. However, some common instruction types require the processor to write back the result to a register and the PC is updated for the next instruction.

The exact details of each step can vary depending on the processor architecture. However, the general sequence of steps is the same for all processors. The instruction execution cycle is a fundamental concept in computer architecture. It is essential for understanding how assembly programs work. By understanding the four steps of the instruction execution cycle, you can gain a deeper understanding of how processors work and how to write efficient assembly code.

3.7 The Clock and the Sequencer

The clock is a device that generates a periodic signal that synchronizes the operations of the CPU. The sequencer is a circuit that controls the order in which instructions are executed.

3.7.1 The Clock

Imagine a metronome keeping time for the processor. The clock sends regular pulses, each tick triggering the next step in the instruction execution cycle. The clock frequency, measured in Hertz (Hz), determines the processing speed. Higher frequency means more instructions executed per second.

3.7.2 The Sequencer

Think of the sequencer as the brain of the instruction execution cycle. It receives the clock pulses and orchestrates the four main stages: Fetch, Decode, Execute and Writeback. The clock and the sequencer work together to ensure that instructions are executed in a precise and orderly fashion.

Additional Points:

- Pipelining can overlap stages of multiple instructions, further increasing performance by utilizing idle processing units.
- Control signals are used to communicate between the clock, sequencer, and functional units, ensuring synchronized execution.
- Different architectures may have additional stages or variations in implementation, but the core functionality remains the same.

Understanding how the clock and sequencer orchestrate the instruction execution cycle is crucial for grasping assembler programs and processor functionality. Remember, this is a general overview, and specific details will vary depending on the architecture you're interested in.

3.8 Conclusion

In this chapter, we have discussed the different types of instructions, the different formats of instructions, and how instructions are executed. Instructions serve as the foundational commands that dictate the operations within computer programs, encompassing arithmetic calculations, logical decisions, flow control, and interactions with input/output devices. Encoded in machine language, these concise sequences of binary digits form the direct communication interface with the computer's central processing unit (CPU). By exploring the various types and formats of instructions, we have gained insights into their role in executing complex operations efficiently. By understanding how instructions work, you can better understand how computers work.

Chapter 4

The processor

4.1 Introduction

The MIPS R3000 is a 32-bit RISC processor that was developed by MIPS Computer Systems in 1988. It was the first commercial processor to implement the MIPS I instruction set architecture. The R3000 was used in a variety of systems, including workstations, servers, and embedded systems.

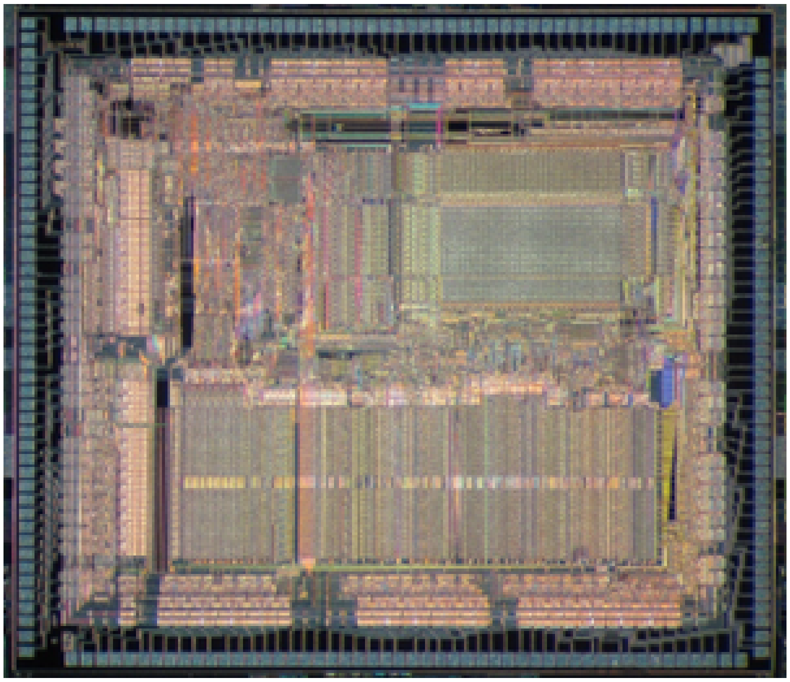


Figure 4.1: The MIPS R3000 [10]

The R3000 is a pipelined processor, which means that it can fetch, decode, execute, and write back instructions in parallel. This allows the R3000 to achieve high performance. The MIPS R3000 was used in a variety of industrial applications such: The MIPS Magnum (workstation), The Sun SPARCstation 1(server), The Nintendo 64(video game console)

4.2 External Architecture of the MIPS R3000 Processor

The external architecture of the MIPS R3000 processor represents what a programmer needs to know in order to program in assembly language. It includes the following:

- **Visible registers:** The R3000 has 32 (32-bit general) purpose registers. These registers are used to store data and instructions.
- **Memory addressing:** The R3000 uses a 32-bit address bus to access memory.
- **Instruction set:** The R3000 instruction set is a RISC instruction set. It includes a variety of instructions for arithmetic, logical, and control operations.
- **Interrupt and exception handling:** The R3000 supports a variety of interrupt and exception handling mechanisms.

4.2.1 Visible registers

A visible register is a register whose value can be read or modified by instructions. Each visible register is a 32-bit register. The processor has two operating modes: user and supervisor. These two operating modes require two categories of registers: unprotected registers and protected registers.

- **Unprotected registers:** are registers that can be accessed by both user-level programs and operating system code. These registers are used to store data and instructions. The following is a list of the unprotected registers in the MIPS R3000 processor:
  - **o** $zero: Always contains the value 0

- $at: Reserved for use by the operating system
- $v0-$v1: Value registers
- $a0-$a3: Argument registers
- $t0-$t9: Temporary registers
- $s0-$s7: Saved registers
- $t8-$t9: Temporary registers
- $gp: Global pointer register
- $sp: Stack pointer register
- $fp: Frame pointer register
- $ra: Return address register

User-level programs can access these registers to store data and instructions. The operating system can also access these registers to pass data and arguments to user-level programs, and to return values from user-level programs. The operating system is responsible for managing the use of unprotected registers. For example, the operating system ensures that user-level programs cannot corrupt the operating system itself by modifying the contents of unprotected registers.

- **Protected registers:** are registers that can only be accessed by operating system code. They are used to control the operation of the processor and the system.

4.2.2 Memory addressing

The MIPS R3000 processor memory is organized as a sequence of 8-bit bytes. This means that each memory location can store one byte of data. Addresses in the MIPS R3000 processor are 32 bits long. This means that the processor can address up to 4GB of memory. Instructions in the MIPS R3000 processor are also 32 bits long. Data can be exchanged between registers and memory in the MIPS R3000 processor using the following instructions:

- Load instructions: Load instructions load data from memory into registers.
- Store instructions: Store instructions store data from registers into memory.

The MIPS R3000 processor can transfer data between registers and memory in three sizes:

- Word: A word is four bytes long.
- Halfword: A halfword is two bytes long.
- Byte: A byte is one byte long.

The addresses of words and instructions must be a multiple of four. The addresses of halfwords must be a multiple of two. If an instruction calculates an address that does not meet this constraint, the processor will generate an exception. The MIPS R3000 processor memory is divided into two segments, identified by the most significant bit of the address:

- User segment: The user segment is accessible in both user mode and supervisor mode.
- System segment: The system segment is only accessible in supervisor mode.

If a user mode instruction tries to access the system segment, the processor will generate an exception.

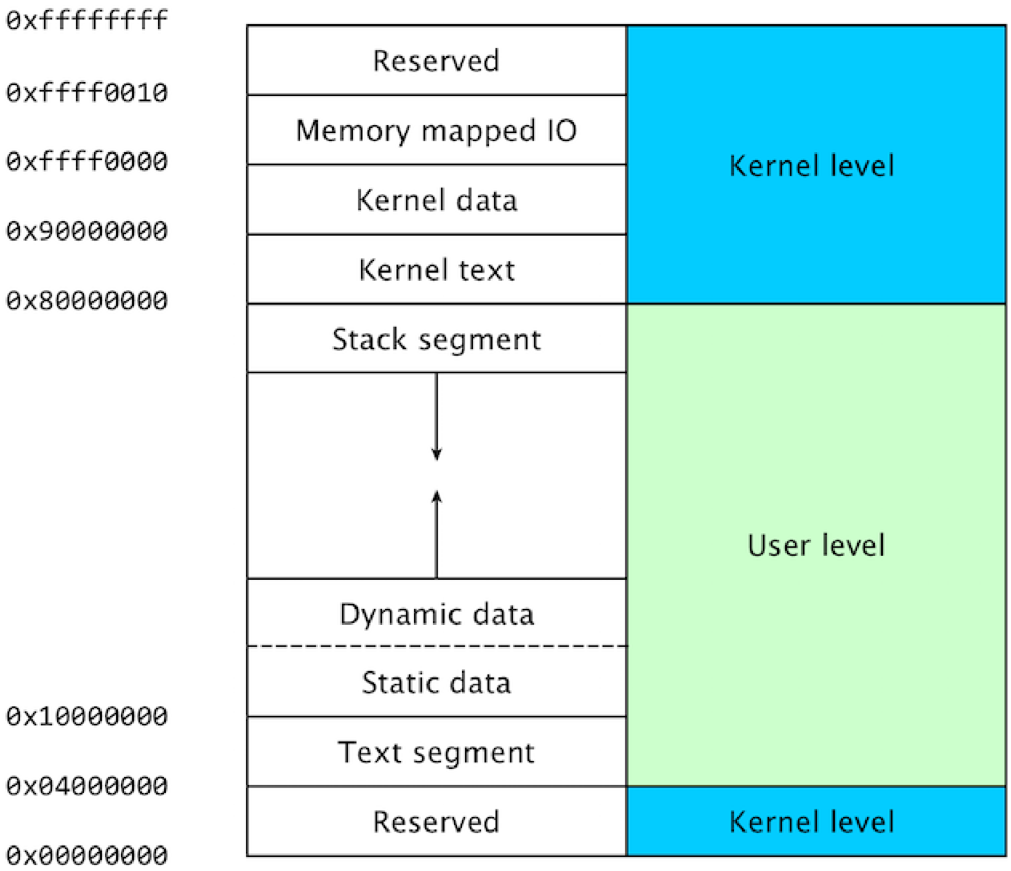


Figure 4.2: Central Memory Architecture in MIPS R3000 [14]

4.2.3 The MIPS R3000 Instruction Set

The MIPS R3000 instruction set is a 32-bit instruction set architecture (ISA) that was introduced in 1988. The R3000 was the second implementation of the MIPS ISA, and it was based on the Berkeley RISC I design. The MIPS R3000 instruction set is a simple and efficient instruction set that is designed for high performance. The instruction set consists of a total of 57 instructions, which are divided into the following categories:

- Arithmetic and logical instructions: These instructions perform arithmetic and logical operations on data.
- Data transfer instructions: These instructions transfer data between registers and memory.
- Control flow instructions: These instructions control the flow of execution of a program.
- System instructions: These instructions perform system-level operations, such as accessing I/O devices.

4.2.3.1 Arithmetic and logical instructions

The arithmetic and logical instructions in the MIPS R3000 instruction set are used to perform arithmetic and logical operations on data. These instructions include:

add: Add the contents of two registers and store the result in a third register.

addu: Add the contents of two registers and store the result in a third register, treating the operands as unsigned integers.

addi: Add a signed immediate value to a register and store the result in the same register.

addiu: Add an unsigned immediate value to a register and store the result in the same register.

sub: Subtract the contents of one register from another register and store the result in a third register.

subu: Subtract the contents of one register from another register and store the result in a third register, treating the operands as unsigned integers.

mult: Multiply the contents of two registers and store the product in the HI and LO registers.

multu: Multiply the contents of two registers and store the product in the HI and LO registers, treating the operands as unsigned integers

div: Divide the contents of one register by another register and store the quotient and remainder in the LO and HI registers, respectively.

divu: Divide the contents of one register by another register and store the quotient and remainder in the LO and HI registers, respectively, treating the operands as unsigned integers.

mfhi: Move the contents of the HI register to another register.

mflo : Move the contents of the LO register to another register.

mthi: Move the contents of a register to the HI register.

mtlo: Move the contents of a register to the LO register.

and : Performs a logical AND operation on the values of the two source registers and stores the result in the destination register.

or : Performs a logical OR operation on the values of the two source registers and stores the result in the destination register.

xor : Performs a logical XOR operation on the values of the two source registers and stores the result in the destination register.

nor : Performs a logical NOT operation on the result of the logical AND operation on the values of the two source registers and stores the result in the destination register.

nand : Performs a logical NOT operation on the result of the logical AND operation on the values of the two source registers and stores the result in the destination register.

slt : Performs a logical comparison on the values of the two source registers and stores the result in the destination register. The result is 1 if the value of source register $rs1 is less than the value of source register $rs2, and 0 otherwise.

sltu: Performs a logical comparison on the values of the two source registers and stores the result in the destination register. The result is 1 if the value of source register $rs1 is less than or equal to the value of source register $rs2, and 0 otherwise.

Examples: Here are some examples of arithmetic and logical instructions in the MIPS R3000 instruction set:

1) add Rd, Rs, Rt: Adds the values of the registers Rs and Rt and stores the result in the register Rd (Rd = Rs + Rt).

2) mult Rs, Rt: Multiplies the values of the registers Rs and Rt and stores the result in the two registers HI and LO. LO receives the lower 32 bits of the result, and HI receives the higher 32 bits of the result.

3) and Rd, Rs, Rt: Performs a bitwise AND operation on the values of the registers Rs and Rt and stores the result in the register Rd.

4) slt Rd, Rs, Rt: compares the value of register Rs to the value of register Rt and stores the result in register Rd. If the value of register Rs is less than the value of register Rt, the result is 1, otherwise the result is 0.

And with values: Code in MIPS R3000

```
li $8 ,0x1025DF14

li $10,0x20124DA0

add $11,8,10 | $11 = 0x30382CB4

and $12,8,10 | $12 = 0x00004D00

mult $8,10

mflo $13 | $13 = 0xAB2C7080

slt $14,8,10 |  $14 = 0x00000001
```

4.2.3.2 Data transfer instructions

Data transfer instructions are used to move data between registers and memory. The MIPS R3000 ISA has a variety of data transfer instructions, including:

Load Instructions: Load instructions load data from memory into a register. The most common load instructions are:

lw : Loads a word (32 bits) from memory into a register.

lh : Loads a halfword (16 bits) from memory into the low-order 16 bits of the register. The sign bit of the loaded value is extended to the remaining 16 high-order bits of the destination register.

lhu : Loads a halfword (16 bits) from memory into the low-order 16 bits of the register, and the remaining 16 high-order bits are padded with zeros.

lb : Loads a byte (8 bits) from memory into the low-order 8 bits of the register. The sign bit of the loaded value is extended to the remaining 24 high-order bits of the destination register.

lbu : Loads a byte (8 bits) from memory into the low-order 8 bits of the register, and the remaining 24 high-order bits are padded with zeros.

Store Instructions: Store instructions store data from a register into memory. The most common store instructions are:

sw : Stores a word (32 bits) from a register into memory.

sh : Stores a halfword (16 bits) from the low-order 16 bits of a register into memory.

sb : Stores a byte (8 bits) from the low-order 8 bits of a register into memory.

Examples: Here are some examples of data transfer instructions:

1) lw Rd, 0(Rs): Loads the word at memory address Rs + 0 into register Rd.

2) sh Rd, -4(Rs): Stores the value in register Rd at memory address Rs - 4.

And with values: Code in MIPS R3000

```
.data
A: .word 0x15D478C5
B: .word 0xF502B1F3
.text
_start:
li $8 ,0x1025DF14
la $20, A
lw $11,4($20) | $11 = 0x F502B1F3
lh $12,4($20) | $12 = 0x FFFFB1F3
lhu $13,4($20) | $13 = 0x 0000B1F3
lb $14,4($20) | $14 = 0x FFFFFFF3
lbu $15,4($20) | $15 = 0x 000000F3
lbu $15,4($20) | $15 = 0x 000000F3
sb $8,4($20) | B = 0x F502B114
sh $8,4($20) | B = 0x F502DF14
sw $8,4($20) | B = 0x 1025DF14
```

4.2.3.3 Control flow instructions

Control flow instructions are used to change the flow of execution of a program. The MIPS R3000 ISA has a variety of control flow instructions, including:

Jump Instructions: Jump instructions transfer control to another part of the program. The most common jump instructions are:

j: Jumps to a label.

jal: Jumps to a label and saves the return address in the $ra register.

Branch Instructions: Branch instructions transfer control to another part of the program based on a condition. The most common branch instructions are:

beq: Branches to a label if two registers are equal.

bne: Branches to a label if two registers are not equal.

bgez: Branch to a label if one register is greater or equal zero

bgtz: Branch to a label if one register is greater than zero

blez: Branch to a label if one register is less or equal zero

bltz: Branch to a label if one register is less than zero

Return Instructions: Return instructions return from a subroutine. The most common return instruction is:

jr: Returns from a subroutine.

Examples: Here are some examples of control flow instructions:
j main: Jumps to the label main.
jal sub1: Jumps to the subroutine sub1 and saves the return address in the $ra register.
beq Rs, Rt, eq: Branches to the label eq if the values in registers Rs and Rt are equal.
bne Rs, Rt, neq: Branches to the label neq if the values in registers Rs and Rt are not equal.
jr $ra: Returns from the current subroutine.

1- Example with If :
If ($8 < 0) then $9 = $9+$10
else $9 = $9-$10
Code in MIPS :

```
bltz $8, label1
sub $9,$9,$10
j label2
label1 : add $9,$9,$10
label2:
```

2- Example with While :

```
While ($8 < 0) do {
$8 = $9+$10
$9 = $9-$10
}
```

Code in MIPS :

```
boucle : Bltz $8, label1
j label2
label1 : add $8,$9,$10
sub $9,$9,$10
j boucle
label2 :
```

4.2.3.4 Shift instructions

Shift instructions are used to manipulate the bits of a register by shifting them a certain number of positions to the left or right. There are two main types of shift instructions: logical shifts and arithmetic shifts. **Logical shifts** fill the vacated bits with zeros. This means that when a bit is shifted out of the register, a zero is inserted in its place. Logical shifts are commonly used to multiply unsigned integers by powers of two. For example, shifting an unsigned integer by one position to the left is equivalent to multiplying it by two. **Arithmetic shifts** fill the vacated bits with the sign bit of the register. This means that if the sign bit is 1, then all of the vacated bits will be filled with 1s. If the sign bit is 0, then all of the vacated bits will be filled with 0s. Arithmetic shifts are commonly used to divide signed integers by powers of two. For example, shifting a signed integer by one position to the right is equivalent to dividing it by two.
The MIPS architecture has four basic shift instructions:

sll : (Shift Left Logical) Shifts the bits of a register to the left by a specified amount.

srl : (Shift Right Logical) Shifts the bits of a register to the right by a specified amount.

sllv : (Shift Left Logical Variable) Shifts the bits of a register to the left by the value in another register.

srlv : (Shift Right Logical Variable) Shifts the bits of a register to the right by the value in another register.

srav : (Shift Right Arithmetic Variable) Shifts the bits of a register to the right by the value in another register.

sra : (Shift Right Arithmetic) Shifts the bits of a register to the right by a specified amount.

And with values: Code in MIPS R3000

```
.text
_start:
li $8 ,0xF025DF14
li $10,0x20124D04
sll $11,$8,8 | $11 = 0x25DF1400
srl $12,$8,8 | $12 = 0x00F025DF
sllv $13,$8,$10 | $13 = 0x025DF140
srlv $14,$8,$10 | $14 = 0x0F025DF1
srav $15,$8,$10 | $15 = 0xFF025DF1
sra $16,$8,8 | $16 = 0xFFF025DF
```

4.2.3.5 System instructions

System instructions are used to interact with the operating system and perform privileged operations. The MIPS R3000 ISA has a variety of system instructions, including:

syscall: Executes a system call.

break: Causes a breakpoint.

trap: Causes a trap.

eret: Returns from an exception.

Examples: Here are some examples of system instructions:
syscall 5, filename, oflag: Opens the file filename with the flags oflag.
break: Causes a breakpoint at the current instruction.
trap 2: Causes a trap with trap number 2.
eret: Returns from the current exception handler.

4.2.4 The instruction encoding

The MIPS R3000 instruction set has three main instruction encoding formats: R, J, and I. These formats determine the arrangement of the instruction's components within the 32-bit instruction word.

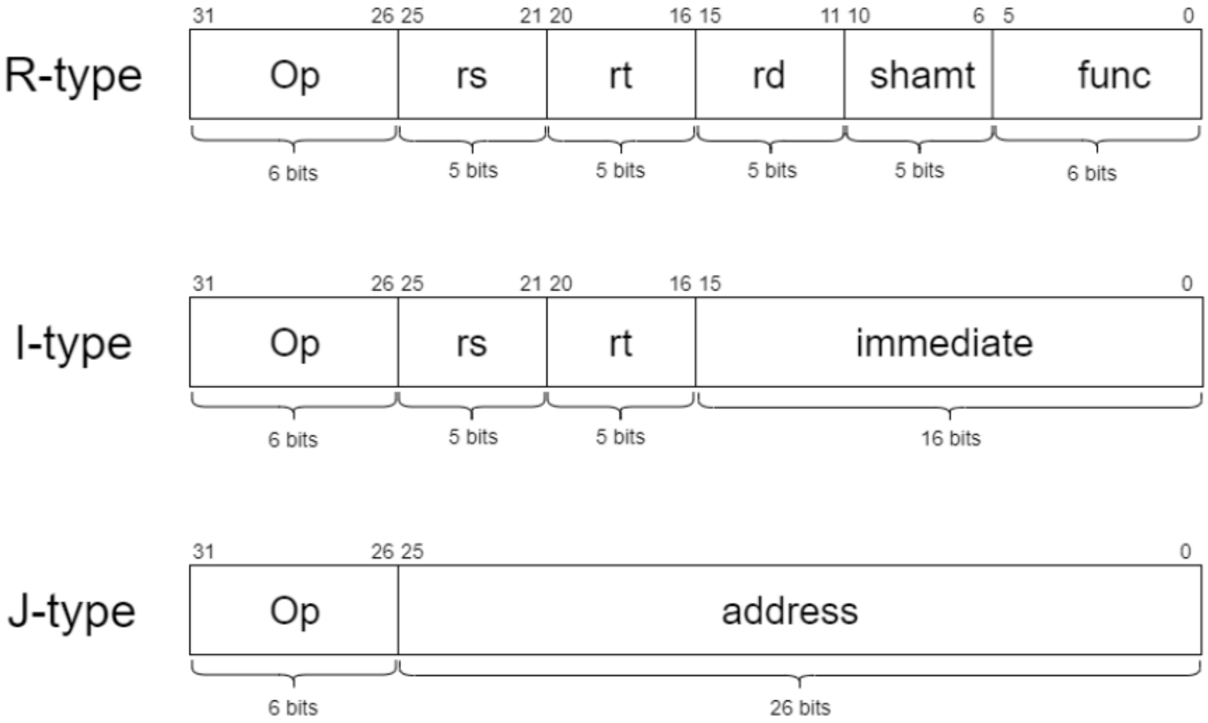
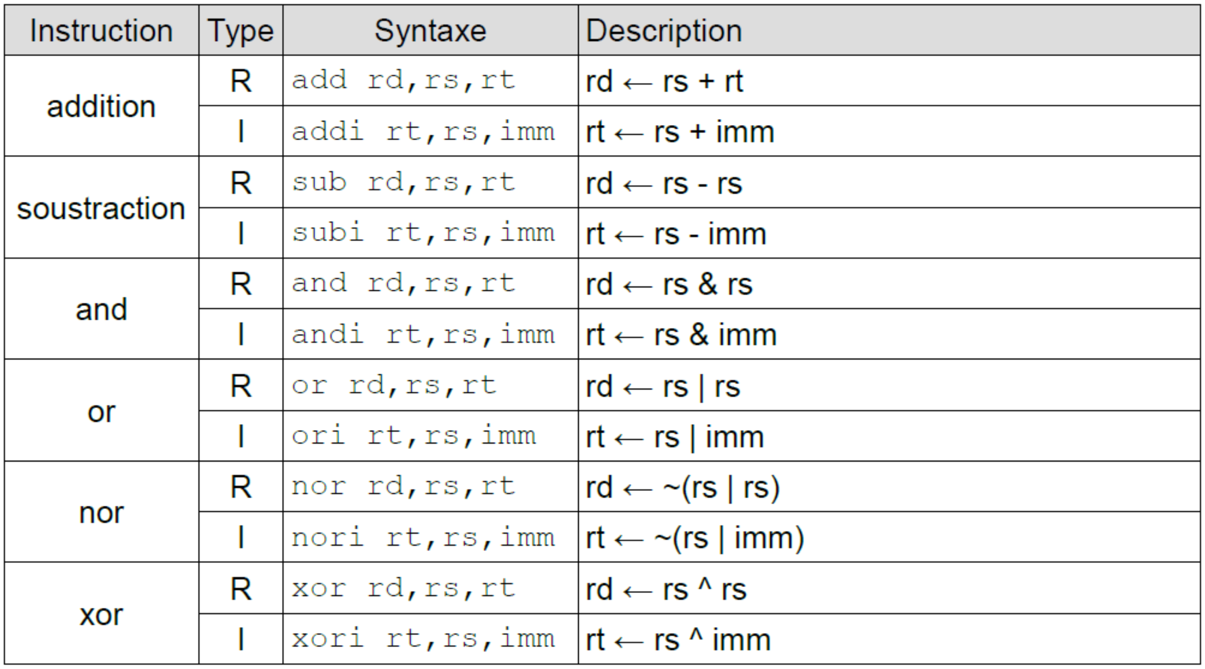


Figure 4.3: Formats: R, J, and I [2]

4.2.4.1 Syntax and format of MIPS R3000 instructions

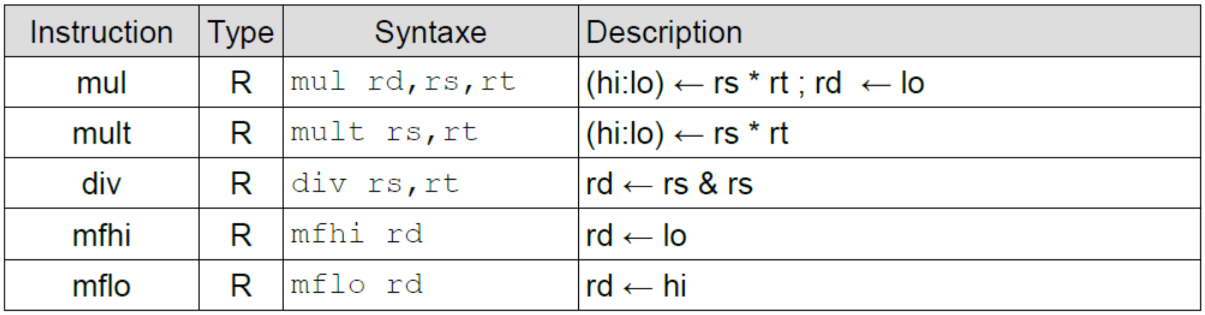
The following figures show all of the MIPS R3000 instructions, grouped by format.

Arithmetic and logical instructions



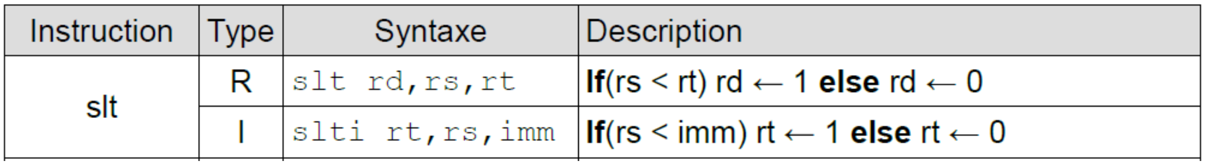
Instruction	Type	Syntaxe	Description
addition	R	add rd,rs,rt	rd ← rs + rt
	I	addi rt,rs,imm	rt ← rs + imm
soustraction	R	sub rd,rs,rt	rd ← rs - rs
	I	subi rt,rs,imm	rt ← rs - imm
and	R	and rd,rs,rt	rd ← rs & rs
	I	andi rt,rs,imm	rt ← rs & imm
or	R	or rd,rs,rt	rd ← rs | rs
	I	ori rt,rs,imm	rt ← rs | imm
nor	R	nor rd,rs,rt	rd ← ~(rs | rs)
	I	nori rt,rs,imm	rt ← ~(rs | imm)
xor	R	xor rd,rs,rt	rd ← rs ^ rs
	I	xori rt,rs,imm	rt ← rs ^ imm

Figure 4.4: Arithmetic and logical instructions (a) [2]



Instruction	Type	Syntaxe	Description
mul	R	mul rd,rs,rt	(hi:lo) ← rs * rt ; rd ← lo
mult	R	mult rs,rt	(hi:lo) ← rs * rt
div	R	div rs,rt	rd ← rs & rs
mfhi	R	mfhi rd	rd ← lo
mflo	R	mflo rd	rd ← hi

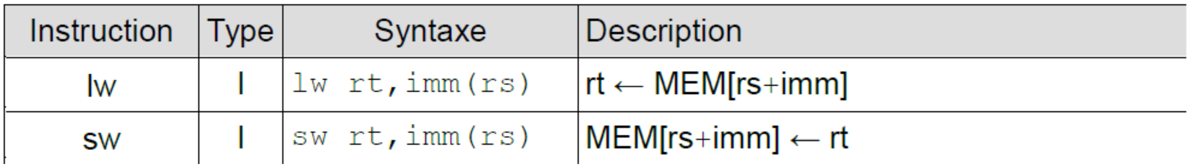
Figure 4.5: Arithmetic and logical instructions (b)[2]



Instruction	Type	Syntaxe	Description
slt	R	`slt rd,rs,rt`	**If**(rs < rt) rd ← 1 **else** rd ← 0
	I	`slti rt,rs,imm`	**If**(rs < imm) rt ← 1 **else** rt ← 0

Figure 4.6: Arithmetic and logical instructions (c) [2]

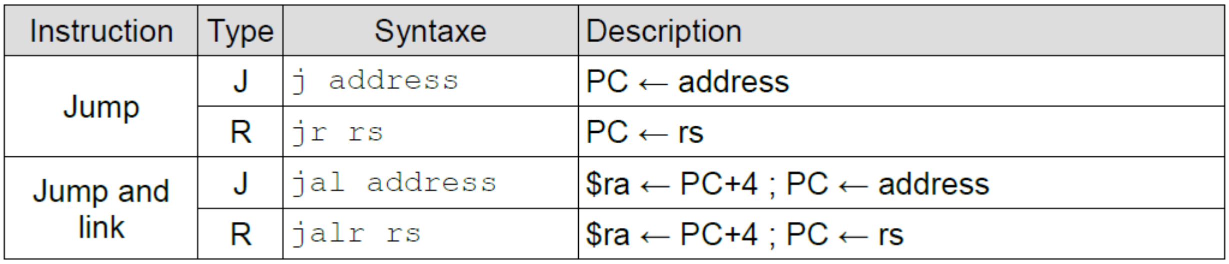
Data transfer instructions



Instruction	Type	Syntaxe	Description
lw	I	`lw rt,imm(rs)`	rt ← MEM[rs+imm]
sw	I	`sw rt,imm(rs)`	MEM[rs+imm] ← rt

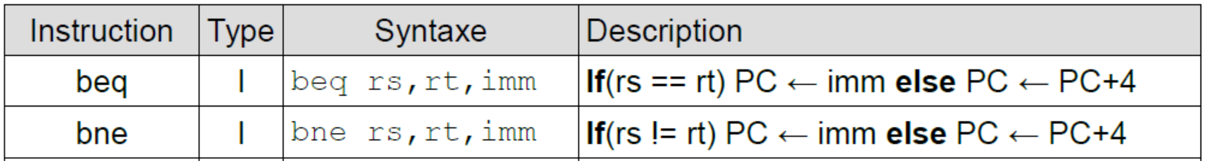
Figure 4.7: Data transfer instructions [2]

Control flow instructions



Instruction	Type	Syntaxe	Description
Jump	J	`j address`	PC ← address
	R	`jr rs`	PC ← rs
Jump and link	J	`jal address`	$ra ← PC+4 ; PC ← address
	R	`jalr rs`	$ra ← PC+4 ; PC ← rs

Figure 4.8: Control flow instructions (a) [2]



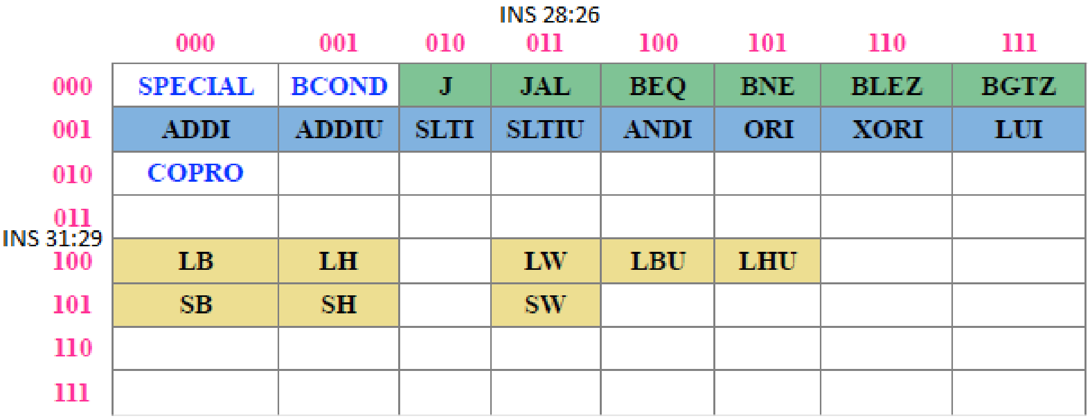
Instruction	Type	Syntaxe	Description
beq	I	`beq rs,rt,imm`	**If**(rs == rt) PC ← imm **else** PC ← PC+4
bne	I	`bne rs,rt,imm`	**If**(rs != rt) PC ← imm **else** PC ← PC+4

Figure 4.9: Control flow instructions (b) [2]

System instructions and shift instructions : All system instructions and shift instructions use the R format

4.2.4.2 Encoding

The type of instruction is primarily determined by the opcode in the instruction operation code field (INS 31:26). Example: the ADDI instruction has the opcode 001000.

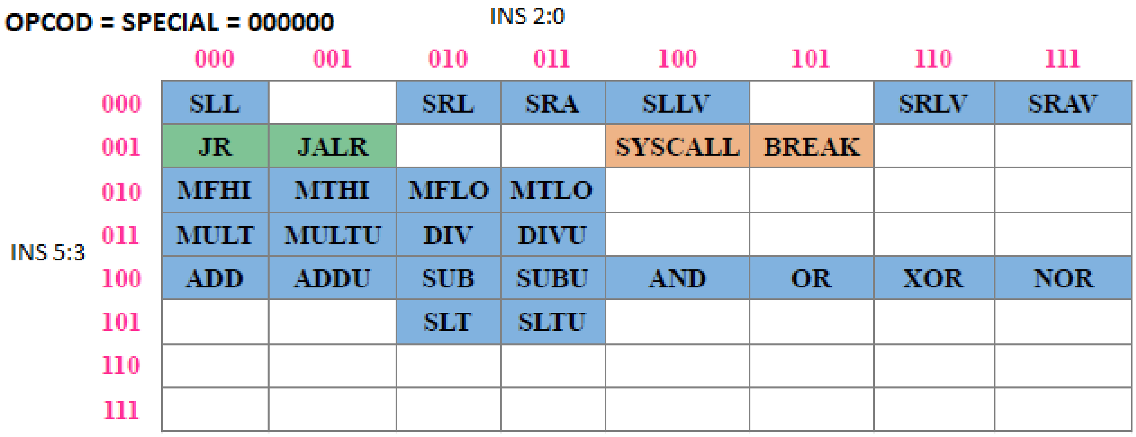


INS 31:29 \ INS 28:26	000	001	010	011	100	101	110	111
000	SPECIAL	BCOND	J	JAL	BEQ	BNE	BLEZ	BGTZ
001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
010	COPRO							
011								
100	LB	LH		LW	LBU	LHU		
101	SB	SH		SW				
110								
111								

Figure 4.10: Tables for encoding (a) [2]

- If the operation code is SPECIAL, you need to look at the 6 least significant bits of the instruction (INS 5:0).
- If the operation code is BCOND, you need to look at bits 20 and 16.

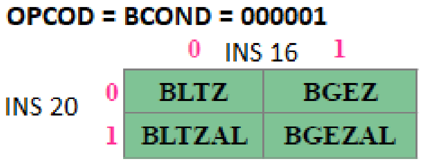
- If the operation code is COPRO, you need to look at bits 25 and 23.



OPCOD = SPECIAL = 000000

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								

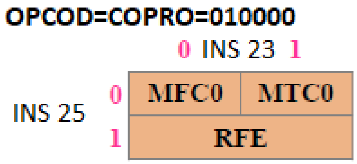
Figure 4.11: Tables for encoding (b) [2]



OPCOD = BCOND = 000001

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

Figure 4.12: Tables for encoding (c) [2]



OPCOD=COPRO=010000

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

Figure 4.13: Tables for encoding (d) [2]

Example of encoding: instruction: nor $10, $5,$11

o **Step 1: Find the syntax and format** :

The syntax of the instruction is nor Rd, Rs, Rt. The format of the instruction is R.

o **Step 2: Find the opcode** :

The opcode for the nor instruction is 000000. The function code for the nor instruction is 100111.

o **Step 3: Make the correspondences** :

The correspondences for the registers are as follows:

Rd = $10 = 01010

Rs = $5 = 00101

Rt = $11 = 01011

o **Step 4: Writing the instruction according to the format** :

0000 0000 1010 1011 0101 0000 0010 0111

o **Step 5: Convert the instruction to hexadecimal** :

0x00AB5027

Example of decoding: instruction in hexadecimal: 0x868C0004

o **Step 1: Convert hexadecimal to Binary** :

Instruction in binary = 10000110100011000000000000000100

o **Step 2: Find the Opcode in 6 high-order bits** :

Opcode = 100001

o **Step 3: Find the syntax and format** :

Syntax = "lh Rt,I(Rs)" Format = "I"

o **Step 4** :

Parse the string according to the format and find the values of Rt, Rs, and I

Rt = 01100 = $12

Rs = 10100 = $20

I = 0000000000000100 = 4

o **Step 5: Format the instruction according to the syntax** :

lh $12,4($20)

4.3 MIPS R3000 programming

4.3.1 Rules of syntax:

- **Structure of a MIPS program:** A MIPS program is typically divided into three sections: text, data, and stack. The text section contains the executable instructions of the program. The data section contains the variables and constants used by the program. The stack section is used to store temporary data and return addresses.

- **Directives:** Directives are instructions to the assembler that provide information about the program or control the assembly process. Directives are written as follows: .directive argument1, argument2, ... The period (.) indicates that the word following it is a directive.

- **Declaration of text, data, and stack sections:** The .text directive is used to declare the start of the text section. The .data directive is used to declare the start of the data section. The .stack directive is used to declare the start of the stack section.

- **Declaration and initialization of variables:** Variables are declared using the .word directive. The .word directive reserves a word of memory for the variable and initializes it to the specified value.

  Example:

  ```
  .data
  var1 : .word 10
  var2 : .word 20
  var3 : .word 30
  ```

  This code declares three variables named var1, var2, and var3. The values of the variables are 10, 20, and 30, respectively. Variables can also be declared using the .space directive. The .space directive reserves a specified number of bytes of memory for the variable.

Example:
.data
Var4 : .space 10
This code declares a variable named var4 that is 10 bytes long.

- **Declaration of data:** Constants are declared using the .asciiz directive. The .asciiz directive reserves a word of memory for the constant and initializes it to the specified string.
  Example:
  .data
  Message : .asciiz "Hello, world!"
  This code declares a constant named message that is the string "Hello, world!".

- **Identifiers (labels):** Identifiers are used to identify labels, variables, and constants. Identifiers must start with a letter or underscore (_) and can contain any combination of letters, digits, and underscores.

- **Types of data** :

  - **Integers:** Integers are whole numbers. Integers in MIPS can be represented in a variety of sizes, including 8 bits, 16 bits, 32 bits, and 64 bits. The .word directive is used to declare a 32-bit integer.
  - **Strings:** Strings are sequences of characters. Strings in MIPS are represented as arrays of characters. The .asciiz directive is used to declare a string.
  - **Reals:** Reals are numbers with a decimal point. Reals in MIPS can be represented in a variety of sizes, including 32 bits and 64 bits. The .float directive is used to declare a 32-bit real.
  - **Registers:** Registers are special-purpose memory locations that are used to store data quickly. MIPS has 32 registers, numbered from 0 to 31.
  - **Comments:** Comments are used to provide human-readable explanations or descriptions within the assembly code. Comments are written as follows:
    ; This is a comment or This is a comment The semicolon (;) indicates that the text following it is a comment and should be ignored by the assembler.

Example:

Here is an example of a MIPS program that prints the message "Hello, world!" to the console:

```
.data
message: .asciiz "Hello, world!"
.text
_start :
la $4, message
li $2, 4
syscall
li $2, 10
syscall
```

This program uses the following directives:

- .data to declare the start of the data section
- .asciiz to declare the constant message
- .text to declare the start of the text section

This program uses the following instructions:

- la to load the address of the constant message into register $4
- li to load the value 4 into register $2.
- ’syscall’ to invoke the system call ’print_string()’, which prints the string pointed to by register

4.3.2 Macro instruction

A macro instruction is a pseudo-instruction that is not part of the machine instruction set, but is accepted by the assembler that translates it into a sequence of several machine instructions. Macro instructions use register $1 if they need to perform an intermediate calculation. Therefore, this register should not be used in programs.

- **Loading an address into a register** :

  Syntax: la $rr, adr

Description: The address considered as an unsigned quantity is loaded into the register.

- **Loading an immediate operand on 32 bits into a register** :
  Syntax: li $rr, imm
  Description: The immediate value is loaded into register $rr.
  Examples:
  la $t0, message
  This instruction loads the address of the variable message into register $t0.
  li $t1, 10
  This instruction loads the value 10 into register $t1.

4.3.3 System calls in MIPS R3000

MIPS R3000 provides a number of system calls that allow programs to interact with the operating system. These system calls are implemented in assembly language and are called by using the syscall instruction.

- **Writing an integer:** To write an integer to the console, you need to do the following:

  - Load the integer to be written into register $4. This is the value that will be written to the console.
  - Load the value 1 into register $2. This is the system call code for writing an integer.
  - Call the syscall() function. This function calls the operating system to write the integer to the console.

  Example:

```
.text
_start:
li $4, 0x00FF00CC # Load the hexadecimal value 0x00FF00CC into $4
ori $2, $0, 1 # Set the ’print_integer’ code in $2
```

```
syscall # Print 0x00FF00CC
```

- **Reading an integer from the console:** To read an integer from the console, you need to do the following:
  - Load the value 5 into register $2. This is the system call code for reading an integer.
  - Call the syscall() function. This function calls the operating system to read the integer from the console.
  - Store the result in register $2. This is the value that was read from the console.

  Example:

  ```
  .text
  _start:
  ori $2, $0, 5 # Set the 'read_integer' code in $2
  syscall # Read an integer
  ```

- **Writing a string :** To write a string to the console, you need to do the following:
  - Load the address of the string to be displayed into register $4. This is the address of the first character in the string.
  - Load the value 4 into register $2. This is the system call code for writing a string.
  - Call the syscall() function. This function calls the operating system to write the string to the console.

  Example:

  ```
  .data
  str: .asciiz "String to write r "
  .text
  _start:
  la $4, str # Load the address of the string into $4
  ```

```
ori $2, $0, 4 # Set the 'print_string' code in $2
syscall # Write the string
```

- **Reading a string from the console:** To read a string from the console, you need the following:
  - Load the value of the pointer into register $4. This is the address of the first character in the buffer.
  - Load the value of the size of the buffer into register $5. This is the maximum number of characters that can be read.
  - Execute the system call number 8. This calls the operating system to read the string from the console into the buffer pointed to by register $4.

  Example:

  ```
  .data
  ch: .space 256
  .text
  _start:
  la $4, ch # Load the pointer into $4
  ori $5, $0, 152 # Load max length into $5
  ori $2, $0, 8 # Set the 'read_string' code in $2
  syscall # Copy the string into the buffer pointed to by $4
  ```

- **Exiting a program:** To exit a program, you need to execute system call number 10. This is done by the following steps:
  - Load the value 10 into register $2.
  - Call the syscall() function.

  Example:

  ```
  ori $2, $0, 10 # Set the 'exit' code in $2
  syscall # exit
  ```

4.4 Conclusion

In this chapter, we delved into the heart of a computer system - the processor. Our exploration began with the MIPS R3000 processor, examining its external architecture, including its visible registers, memory addressing mechanisms, instruction set, and instruction encoding. We then delved into the world of MIPS R3000 programming, understanding the rules of syntax, macro instructions, and system calls. Through this journey, we gained a deeper appreciation for the processor's role in executing instructions and managing data within a computer system.

This chapter serves as a foundation for further exploration of processor design and architecture. As you delve deeper, you'll discover more complex architectures with advanced features like pipelining and caching. Remember, the processor remains the fundamental building block of modern computing, and understanding its workings empowers you to better grasp the heart of a computer system.

Chapter 5

Special Instructions

5.1 Introduction

This chapter examines special instructions in computer architecture, focusing on the MIPS R3000 processor.

5.2 Interrupt Handling

5.2.1 Definition and Types of Interrupts

An interrupt is an event that suspends the execution of the current program to handle an urgent external event. The processor can handle two types of interrupts:

Internal Interrupts:

- Arithmetic overflow
- Page fault
- Stack overflow
- Division by zero

External Interrupts:

- Signal from an I/O device (device ready, service request)
- Timer
- Bus error

5.2.2 Interrupt Handling Mechanism

1. **Save the context of the current program:** **o** Program registers (PC, PSR, GPR)

   o Processor control registers (EPC, Cause, Status)

2. **Determine the source of the interrupt:** **o** Interrupt vector (exception vector for internal interrupts)

   o Each interrupt has its own ID number and vector location

3. **Load the interrupt handler:** **o** Address of the interrupt handler stored in the interrupt vector

   o The interrupt handler is an Interrupt Service Routine (ISR)

4. **Execute the interrupt handler:** **o** Process the event that caused the interrupt

   o Read data from the device

   o Send data to the device

   o Acknowledge the interrupt

5. **Restore the context of the current program:** **o** Restore saved registers

   o Resume program execution at the next instruction

Example:

A user presses a key on the keyboard.

1. The keyboard controller generates an interrupt.
2. The processor suspends the execution of the current program and saves its context.

3. The processor determines the source of the interrupt (keyboard) by consulting the interrupt vector.

4. The processor loads the keyboard interrupt handler.

5. The interrupt handler reads the key code and stores it in memory.

6. The interrupt handler sends an end-of-interrupt signal to the processor.

7. The processor restores the context of the current program and resumes program execution.

5.3 Input/Output (I/O)

5.3.1 I/O Instructions

The MIPS R3000 has a set of instructions dedicated to I/O operations: (As seen in Chapter 4)

o **lw** (Load Word): Load a word from memory to a register

o **sw** (Store Word): Store a word from a register to memory

o **lb** (Load Byte): Load a byte from memory to a register

o **sb** (Store Byte): Store a byte from a register to memory

o **lhu** (Load Halfword Unsigned): Load an unsigned halfword from memory to a register

o **sh** (Store Halfword): Store a halfword from a register to memory

5.3.2 Types of I/O Access

- Programmed I/O:
  - **o** The programmer controls the data transfer.
  - **o** Specific I/O instructions for each device.

o Example: lw $t0, 0x1000 (read a word at address 0x1000 into register $t0)

- Interrupt-driven I/O:

  o The device signals the end of an operation with an interrupt.

  o The interrupt handler handles the data transfer.

  o Reduces the processor load.

  o Example: Keyboard end-of-read interrupt

5.4 System Instructions

The MIPS R3000 has a set of instructions for managing system tasks: (As seen in Chapter 4)

- Memory Management:

  o lui (Load Upper Immediate): Load the upper half of a register

  o ori (Or Immediate): Perform a logical OR with a register and an immediate value

  o sw (Store Word): Write a word from a register to memory

- Control Flow:

  o beq (Branch on Equal): Branch to a target address if two registers are equal

  o bne (Branch on Not Equal): Branch to a target address if two registers are not equal

  o j (Jump): Jump to a target address

5.5 Conclusion

In this chapter, we explored the concept of special instructions in computer architecture, with a focus on the MIPS R3000 processor. We covered the following topics:

- **Interrupts:** We discussed the different types of interrupts, the mechanism for handling them, and an example of an interrupt scenario.

- **Input/Output (I/O):** We examined the I/O instructions available in the MIPS R3000, as well as the two main types of I/O access: programmed I/O and interrupt-driven I/O.

- **System Instructions:** We explored the various system instructions available in the MIPS R3000, including those for memory management, control flow, and processor control.

Chapter 6

Practice exercises

6.1 Series 1: Reminders and Revisions

Exercise 1:

Conversion Exercises between Bases 10, 2, and 16 Here are some exercises to practice converting numbers between bases 10, 2 (binary), and 16 (hexadecimal):

Level 1: Decimal to Binary and Hexadecimal

1. Convert the following decimal numbers to binary and hexadecimal:

- o 25
- o 144
- o 777
- o 1023

2. Choose any two random decimal numbers between 1 and 1000 and convert them to binary and hexadecimal. Share your results with a classmate and compare your answers.

Level 2: Binary to Decimal and Hexadecimal

1. Convert the following binary numbers to decimal and hexadecimal:

- o 10101
- o 1110010
- o 10011111

2. Generate your own binary numbers using a random number generator or coin flips (heads = 1, tails = 0). Convert them to decimal and hexadecimal. Then, challenge a friend to do the same and compare your results.

Level 3: Hexadecimal to Decimal and Binary

1. Convert the following hexadecimal numbers to decimal and binary:

- o A3
- o F9C
- o 10EC

2. Use an online hexadecimal converter or a calculator with hexadecimal functionality to find two random hexadecimal numbers between 10 and FF. Convert them to decimal and binary. Share your findings with a group and discuss the different representations of the same number.

Exercise 2 : Arithmetic Operations with Binary and Hexadecimal

Level 0: Complete the following tables:
Table 1 : A+ B

A	B	Result	Carry

Table 2 : A- B

A	B	Result	Carry

Level 1: Warm-up

1. Binary Addition: Add the following binary numbers:

- 1101 + 1010 = ?
- 10011 + 1100 = ?

2. Hexadecimal Addition: Add the following hexadecimal numbers:

- A3 + 2F = ?
- 10EC + C1D = ?

Level 2: Up the Game

1. Binary Subtraction: Subtract the following binary numbers:

- 1110 - 1011 = ?
- 100101 - 10001 = ?

2. Hexadecimal Subtraction: Subtract the following hexadecimal numbers:

- B5 - 6A = ?
- 1F2E - D9C = ?

Level 3: Master Challenge

1. Binary Multiplication: Multiply the following binary numbers:

- 101 x 11 = ?
- 1101 x 100 = ?

2. Hexadecimal Multiplication: Multiply the following hexadecimal numbers:

- A x 5 = ?
- 1C x 2D = ?

Exercise 3 : Logic Operations

Complete the following truth table

A	B	A AND B	A OR B	A NAND B	A NOR B	A XOR B

Exercise 4 : Tow's Complement

Recall : The two's complement of a binary number is the number obtained by :

- **Step 1:** Invert all the bits of the binary number (flip 0s to 1s and vice versa). (1's Complement)
- **Step 2:** Add 1 to the resulting inverted number (carry over if necessary).

1 Find the two's complement of the following binary numbers:

o 1001

o 1101

o 1011

2 Convert the following decimal numbers to two's complement:

o -1

o -2

o -3

6.2 Series 2: Course Questions

Exercise 1:

Draw the two architectures seen in chapter 2 and give the main difference between the two architectures.

Exercise 2: What is the meaning of the following acronyms:

1) CPU: ....

2) UAL: ....

3) RAM: ....

4) ROM: ....

5) DMA: ....

6) USB: ....

Exercise 3: True or false

1. Magnetic and optical disks constitute the main memory.

□ True

□ False

2. Random access memory is the place where programs are stored.

□ True

□ False

3. The execution of an instruction goes through the execution phase and then the fetch phase.

□ True

□ False

4. The program counter stores the instruction that is currently being executed.

□ True

□ False

5. RAM is the storage location for programs.

□ True

□ False

6. The instruction register stores the result of the currently executing instruction.

□ True

□ False

7. The Accumulator register is located in the CU.

□ True

□ False

8. Read-only memory (ROM) is the storage location for currently running programs and data used

□ True

□ False

9. The accumulator is a register in the arithmetic and logic unit.

□ True

□ False

10. PC is the Program Counter Register

□ True

□ False

6.3 Series 3: application exercises

Exercise 1: Give the machine code (hexadecimal form) of the following MIPS instructions:

1. add $20, $10, $8

2. sub $21, $19, $6

3. addi $18, $13, 8

4. or $10, $15, $9

5. and $16, $15, $7

6. ori $13, $24, 15

7. srav $10, $8, $13

8. sllv $15, $13, $17

9. mfco $19, $15

10. sltu $21, $14, $3

Exercise 2: Decode the following MIPS instructions:

1. 0x80137800

2. 0x01C3A82B

3. 0x022D7804

4. 0x01A85007

5. 0x370D000F

6. 0x01E78024

7. 0x01E95025

8. 0x21B20008

9. 0x0266A822

10. 0x0148A020

Exercise 3: Execute the following instructions: (Given: $8 = 0x054F3CB2, $10 = 0xF502B1F3)

1. add $20, $10, $8

2. sub $21, $10, $8

3. addi $18, $10, 8

4. or $10, $10, $8

5. and $16, $10, $8

6. ori $13, $10, 8

7. srav $10, $10, $8

8. sllv $15, $10, $8

9. div $10, $8

10. sltu $10, $8

Exercise 4: In a memory, the byte order used is Little Endian, given $10 = 0x0FEA001A, $8 = 0x5F1A2238. Execute the following instructions:

0x0FEA001A	AA
	F3
	33
	A0
	50
	31
	EF
0x0FEA0021	00

1. lhu $11, 2($10)

2. lh $11, 2($10)

3. lbu $11, 6($10)

4. lb $11, 6($10)

5. lw $11, 2($10)

6. sb $8, 1($10)

7. sh $8, 2($10)

8. sw $8,4($10)

Exercise 5: Using the memory of exercise 4, give the addressing mode for each instruction: RI = 0x0FEA001F et BR = 0x0FEA001C

a. Load 0x0FEA001A $\Longrightarrow$ acc = AA

b. Load 0x1A $\Longrightarrow$ acc = 1A

c. Load 0x0FEA001A $\Longrightarrow$ acc = 31

d. Load 0x0FEA001A $\Longrightarrow$ acc = 33

Exercise 6: For each of the following instructions, write the corresponding MIPS code

```
Q1 :  If $8 < $10 then $13 = $11 * $12 ;
      end_if ;

Q2 :  If $8 < $10 then $13 = $11 * $12 ;
      else $9 = $9 - 1 ;
      end_if ;

Q3 :  While $8 <> $10 do
      $13 = $11 * $12 ;
      $14 = $11 / $12 ;
      End_while ;

Q4 :  If $8 < $10 then
      While $8 <> $10 do
      $13 = $11 * $12 ;
```

```
    $14 = $11 / $12 ;
    End_while ;
    else $9 = $9 - 1 ;
    end_if ;
```

Q5 :

```
While $8 <> $10 do
    If $8 < $10 then $13 = $11 * $12 ;
    else $9 = $9 - 1 ;
    end_if ;
    End_while ;
```

Exercise 7: Assume a MIPS processor that executes the following program:

```
xor $t0,$t0,$t0
addi t1,$t0,0x50
a: lw $t2,0($t1)
beq $t2,$zero,c
slt $t3,$t0,$t2
beq $t3,$zero,b
lw $t0,0($t1)
b: addi $t1,$t1,4
j a
c: sw \$t0,0(\$t1)
```

Q. Explain the function of the program ?

Exercise 8: Consider the following C program snippet

```
int x, y ;
 main () {
while (x !=y) {
                if (x > y) x=x-y ;
                else y=y-x ;
                }
          }
```

Q. Write the corresponding assembly program using MIPS instructions

Exercise 9: Give the sequence of MIPS instructions to perform the following actions:

Q1. Using the system call capabilities for input and output, write a program that repeatedly reads an integer from the keyboard and adds it to the previously read integers. The program stops when the integer read is 0, and then displays the message "Sum = ", followed by the result.

Q2. Write a MIPS program that displays the elements of an array, indicating the size and elements.

Q3. Set the 8000 bytes starting at memory address 014FA1B2 to zero (assuming that register R2 contains 014FA1B2)

Bibliography

[1] YESSAD Samira et BELKHIRI Louiza, Cours et Exercices Architecture des Ordinateurs, Plateforme E-learning de l'université de Bejaia., 2020.

[2] Kara Abdelaziz. *Cours de Computer Architecture*. Computer Science Department, Faculty of Sciences, University of Setif 1, 2020.

[3] en.wikipedia.org. *https://en.wikipedia.org/wiki/Harvard_ architecture*. 2024.

[4] en.wikipedia.org. *https://en.wikipedia.org/wiki/Von$_N$eumann_ architecture.*2024.

[5] John L. Hennessy and David A. Patterson. *Introduction to computer architecture*. Morgan Kaufmann, 2017.

[6] https://shawngraham.io. *https://shawngraham.io/?p=1147*. 2024.

[7] Andrew J. Smith. A survey of emerging memory technologies. *IEEE Transactions on Computer Architecture*, 45(5):1341–1365, 2018.

[8] William Stallings. *Computer organization and architecture: a modern synthesis*. Pearson, 2017.

[9] Andrew S. Tanenbaum and Herbert Bos. *Operating systems: three easy pieces*. Pearson Education, 2015.

[10] wikimedia.org. *https://upload.wikimedia.org/wikipedia/6/60/MIPSR3000Adie.JPG*. 2024.

[11] www.btechsmartclass.com. *https://www.btechsmartclass.com/c_ programming/C-Computer-Languages.html*. 2024.

[12] www.codingninjas.com. *https://www.codingninjas.com/studio/library/memory-hierarchy*. 2024.

[13] www.geeksforgeeks.org. *https://www.geeksforgeeks.org/computer-organization-hardwired-vs-micro-programmed-control-unit/*. 2024.

[14] www.it.uu.se. *https://www.it.uu.se/education/course/homepage/os/vt18/module-0/mips-and-mars/mips-memory-layout/*. 2024.