Karnataka State Open University
Mukthagangotri, Mysore-570006

M.Sc. Computer Science

First Semester

COMPUTER ARCHITECTURE

COURSE – MCSDSE-1.5

CREDIT PAGE

Programme Name: MSc-Computer Science **Year/Semester: I** Semester Block **No:1-3**

Course Name: Computer Architecture **Credit:** 3 **Unit No:** 1-12

Course Design Expert Committee

Dr. Vidyashankar S. Chairman
Vice Chancellor,
Karnataka State Open University,
Mukthagangotri, Mysuru-570006.

Dr. Ashok Kamble Member
Dean (Academic),
Karnataka State Open University,
Mukthagangotri, Mysuru-570 006.

Editorial Committee

Dr. D M Mahesha MCA.,PhD Chairman
BOS Chairman,
Assistant Professor & Programme co-ordinator(PG)
DoS&R in Computer Science,
Karnataka State Open University, Mysuru-570 006.

Smt, Suneetha MSc.,(PhD) Member Convener
Dept Chairperson & Programme co-ordinator (UG)
DoS&R in Computer Science,
Karnataka State Open University, Mysuru-570 006.

Dr Bhavya D.N., MTech., PhD Member
Assistant Professor & Programme co-ordinator(UG)
DoS&R in Computer Science,
Karnataka State Open University, Mysuru-570 006.

Dr. Ashoka S B., MSc.PhD Member
External Subject Expert,
Assistant Professor,
DoS&R in Computer Science,
Maharani's Cluster University, palace Road Bangalore-01

Name of Course Writer	No of Blocks	No of Units	Name of Course Editor	No of Units
Dr. Bhavya D.N., BE., M.Tech., Ph.D., Assistant Professor DoS&R in Computer Science, Karnataka State Open University, Mysuru-06	Block-1	1-2	**Dr. Sumati Ramakrishna Gowda.,** BE (CS & E)., MSc(IT)., MPhil(CS)., Ph.D., Assistant Professor DoS&R in Computer Science,Karnataka State Open University, Mysuru-06	1-2
Dr.Basappa B Kodada BE., MTech., PhD Associate professor Department of Computer Science,AJ Institute of Engineering and Technology Mangalore	Block-1-3	3-12	**Sri. Somashekar BM** BE., MTech., (PhD) Assistant Professor, Department of Information Science MIT Naguvanhalli post, SR Patna Taluk571438	3-12

Developed by the Department of Studies and Research in Computer Science, under the guidance of Dean (Academic), KSOU, Mysuru.
Karnataka State Open University, February-2022.

Printed and Published on behalf of Karnataka State Open University, Mysore-570 006 by the **Registrar (Administration)-2022**

BLOCK 1: CONTENT

preface

This material is prepared to give an overview of the computer architecture for the first semester course in computer science curricula. It is suitable for the both hardware and software oriented students. To study the design details of hardware technology and the architecture of various components of a components of a computer system and its effect on programming. To understand the architecture on the basis of implementation of simple components. The whole material is organized into 2 blocks each with four units. Each unit lists the objectives of study along with the relevant questions and suggested reading to better understand the concepts.

Block 1 named Overview of von Newman Architecture structure of computers introduces entire picture of interconnected to form a complete control unit of computer system. Memory and i/o devices and their interfacing to the CPU measuring and reporting performance; CISC and RISC processors. Pipelining. This block presents the Basic concepts of pipelining, data hazards, control hazards, and structural hazards; Finally, the Techniques for overcoming or reducing the effects of various hazards. Hierarchical Memory Technology are also explained.

Block 2 Introduces the Inclusion, and Coherence and locality properties as well as Cache memory organizations, Techniques for reducing cache misses. This block we also discussed about the Virtual memory organization, mapping and management techniques, memory replacement policies. Instruction-level parallelism: Concepts of instruction-level parallelism (ILP), Techniques for increasing ILP, finally Superscalar, super-pipelined and VLIW processor architectures; Vector and Array Processors are also explained.

Block 3 This block named Principles: Instruction types, Compound, Vector loops, Chaining, Array processor structure and algorithms, Case studies of contemporary microprocessors.

The Block also covers Multiprocessor Architecture: Centralized shared-memory architecture, synchronization, memory consistency, interconnection networks; It also covers the Distributed shared-memory architecture, Cluster computers are also explained.

UNIT – 1 : BASICS OF COMPUTER ARCHITECTURE

Structure

1.0 OBJECTIVES

After going through this lesson you will be able to

- Defining computer architecture
- Describe von Neumann architecture
- Elucidate instruction set architecture
- Discuss arithmetic and logic unit
- Explain control unit

1.1 INTRODUCTION

In this unit, we will discuss a few levels of abstraction to define the architecture of a computer. The *architecture* is the programmer's view of a computer. It is defined by the instruction set (language) and operand locations (registers and memory). Many different architectures exist, such as ARM, x86, MIPS, SPARC, and PowerPC. In this unit explained von-neumann architecture.

1.2 BASICS OF COMPUTER ARCHITECTURE

Computer architecture can be defined as a set of rules and methods that describe the functionality, management and implementation of computers. To be precise, it is nothing but rules by which a system performs and operates.

Sub-divisions

Computer Architecture can be divided into mainly three categories, which are as follows −

- **Instruction set Architecture or ISA** − Whenever an instruction is given to processor, its role is to read and act accordingly. It allocates memory to instructions and also acts upon memory address mode (Direct Addressing mode or Indirect Addressing mode).
- **Micro Architecture** − It describes how a particular processor will handle and implement instructions from ISA.
- **System design** − It includes the other entire hardware component within the system such as virtualization, multiprocessing.

Role of computer Architecture

The main role of Computer Architecture is to balance the performance, efficiency, cost and reliability of a computer system.

For Example − Instruction set architecture (ISA) acts as a bridge between computer's software and hardware. It works as a programmer's view of a machine.

Computers can only understand binary language (i.e., 0, 1) and users understand high level language (i.e., if else, while, conditions, etc). So to communicate between user and computer, Instruction set Architecture plays a major role here, translating high level language to binary language.

Structure

Let us see the example structure of Computer Architecture as given below.

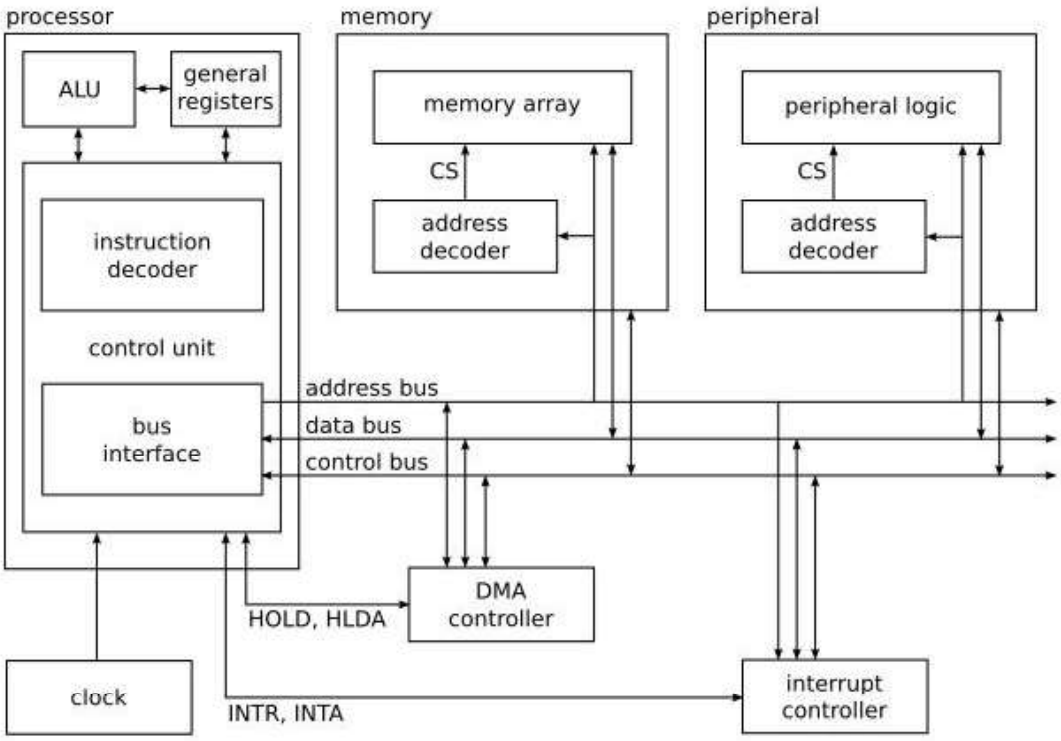
Generally, computer architecture consists of the following −

- Processor
- Memory

- Peripherals

All the above parts are connected with the help of system bus, which consists of address bus, data bus and control bus.

The diagram given below depicts the computer architecture −

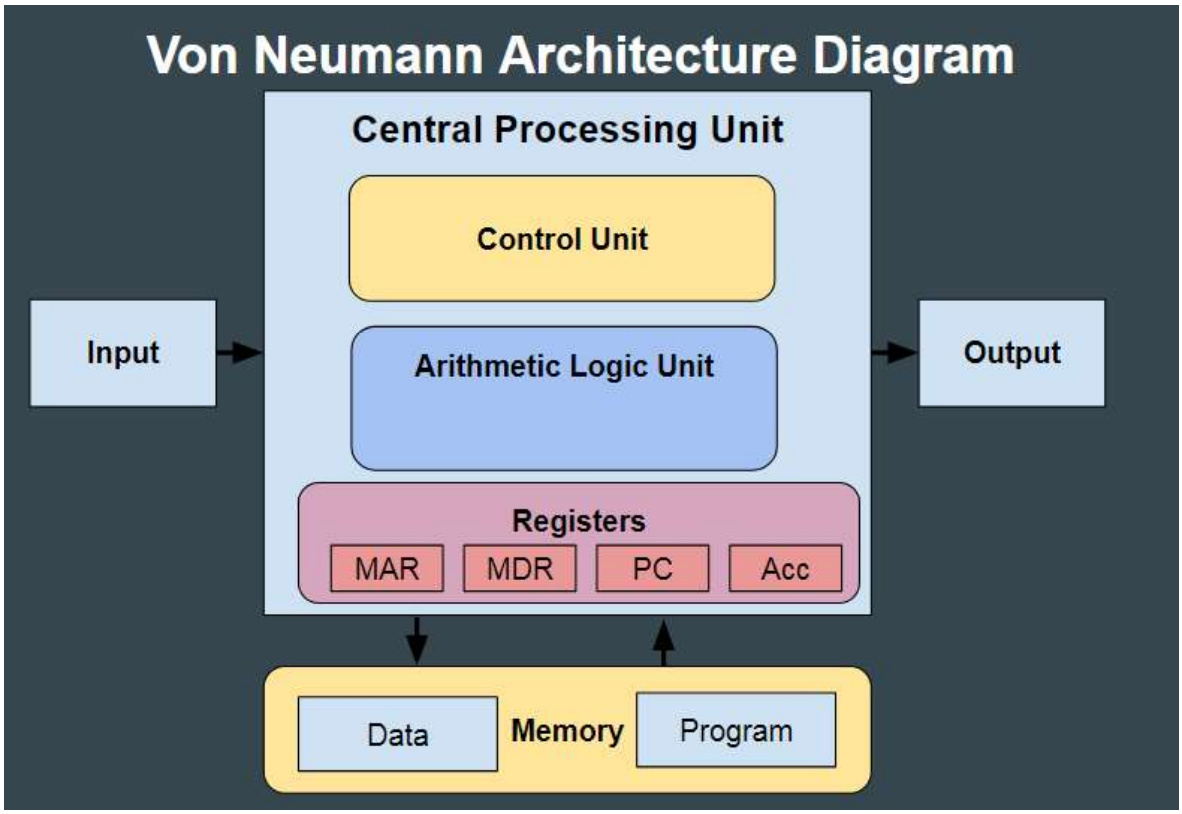


1.3 OVERVIEW OF VON NEUMANN ARCHITECTURE

The definition of Von Neuman Architecture can be originally referred to the specific proposed architecture of a computer's architecture, In the year 1945 John von Neumann as written. The definition has since evolved to refer to specific types of computers. One of the primary characteristics of these computers is that their data operations and instrument fetch processes will occur at the same time – something that was previously impossible until the implementation of the von Neumann architecture.

The von Neumann architecture—the fundamental architecture upon which nearly all digital computers have been based—has a number of characteristics that have had an immense impact

on the most popular programming languages. These characteristics include a single, centralized control, housed in the central processing unit, and a separate storage area, primary memory, which can contain both instructions and data. The instructions are executed by the CPU, and so they must be brought into the CPU from the primary memory. The CPU also houses the unit that performs operations on operands, the arithmetic and logic unit (ALU), and so data must be fetched from primary memory and brought into the CPU in order to be acted upon. The primary memory has a built-in addressing mechanism, so that the CPU can refer to the addresses of instructions and operands. Finally, the CPU contains a register bank that constitutes a kind of "scratch pad" where intermediate results can be stored and consulted with greater speed than could primary memory.



1.4 INSTRUCTION SET ARCHITECTURE

The **I**nstruction **S**et **A**rchitecture is an abstract model of a computer, responsible for the definition of the Data Types, Registers, and the components that manage the Main Memory and the fundamental operation required in a generic computer system, such as Integer Arithmetics.

The ISA designates the behavior of the machine code in the developed computer, providing binary compatibility between different implementations and enabling multiple implementations of the same IS differing in physical size, performance, and manufacturing cost to run the same machine code.

This ISA characteristic makes it possible for, lower-cost computers, lower-performance to be replaced with higher-performance, higher-cost machines without requiring software refactoring.

Seven Dimensions of an ISA

The ISA document is the boundary between the hardware and software implementation in the computer system. The ISA, as defined by Computer Architecture: A Quantitative Approach is created within seven dimensions is given below:

- **Class of ISA —**the majority of the ISAs are categorized to a General-Purpose Register Architectures that implement the operands as either registers or memory locations within the system. General-Purpose Register Architectures are further classified into Register Memory ISAs, which access memory through a large set of instructions (80x86 ISA popular in the development of Personal and Desktop computers) and Load-Store ISAs(made popular by the ARM chips), where memory access is restricted to the load and store instructions;

- **Memory addressing —** The Memory addressing dimension of an ISA tells how memory operands are accessed within memory. Virtually all computers should use byte addressing to access memory operands. Specific architectures, such as ARM and MIPS, require data alignment while others, like the popular 80x86, although not requiring data alignment provide faster memory access when operands are aligned;

- **Addressing modes —** The ISA addressing modes indicate how operands are addressed within the instruction. The MIPS ISA includes three addressing modes: **Register**, **Displacement**, and **Immediate**(for constants), where a constant offset is added to a register to form the memory address. The ARM ISA includes all the MIPS addressing modes and adds the PC-relative addressing, the sum of two registers, and the sum of two registers where one register is multiplied by the size of the operand in bytes. This architecture also provides autodecrement and autoincrement addressing, for sequential access of memory objects. The 80x86 supports the MIPS addressing modes plus three

variations of two registers (based indexed with displacement), displacement: no register (absolute), and two registers where one register is multiplied by the size of the operand in bytes (based with scaled index and displacement);

- **Types and sizes of operands —** The ISAs defines the supported operand sizes and types by the circuitry of the underlying computer. The most popular ISA like the ARM,80x86, and MIPS support 8-bit, 16-bit, 32-bit, and 64-bit integer formats and IEEE 754 floating point in 32-bit (single precision) and 64-bit(double precision) formats;

- **Operations —**The ISA includes the definition of the supported operations of the architecture, which are divided into Arithmetic and Logical, Data Transfer, Control and Floating point;

- **Control flow instructions —** The ISA defines the supported unconditional jumps, conditional branches, procedure calls, and returns. These Control Flow operations use PC-relative addressing, where the branch address is specified by an address field that is added to the PC. The MIPS ISA conditional branches (BE, BNE, etc.) test the contents of registers, while the 80x86 and ARM ISAs conditional branches test condition code bits set as side effects of arithmetic/logic operations, such as the Carry bit from the ALU(Arithmetic Logic Unit).

1.5 ARITHMETIC AND LOGIC UNIT

Arithmetic Logic Unit (ALU): A ALU is the sub unit within a computer's central processing unit. The full form of ALU is **Arithmetic Logic Unit** , takes the data from Memory registers; **ALU** contains the logical circuit to perform mathematical operations like subtraction, addition, multiplication, division, logical operations and logical shifts on the values held in the processors registers or its accumulator.

It is the size of the word that the **ALU** can handle which, more than any other measure, determines the word-size of a processor: that is, a *32-bit processor* is one with a *32-bit ALU*.

After processing the instructions the result will store in Accumulator. Control unit generates control signals to ALU to perform specific operations. The accumulator is used as by default register for storing data. It is 16-bit register. The simplest sort of ALU performs only addition, Boolean logic (including the NOT or complement operation) and shifts a word one bit to the right or left, all other arithmetic operations being synthesized from sequences of these primitive

operations. For example, subtraction is performed as complement-add multiplication by a power of two by shifting, division by repeated subtraction. However, there is an increasing tendency in modern processors to implement extra arithmetic functions in hardware, such as dedicated multiplier or divider units.

The ALU might once have been considered the very core of the computer in the sense that it alone actually performed calculations. However, in modern SUPER SCALAR processor architectures this is no longer true, as there are typically several different ALUs in each of several separate integer and floating-point units. An ALU may be required to perform not only those calculations required by a user program but also many internal calculations required by the processor itself, for example to derive addresses for instructions that employ different ADDRESSING MODES, say by adding an offset to a base address. Once again, however, in modern architectures there is a tendency to distribute this work into a separate load/store unit.

The three fundamental attributes of an ALU are its operands and results, functional organization, and algorithms.

The operands and results of the ALU are machine words of two kinds: arithmetic words, which represent numerical values in digital form, and logic words, which represent arbitrary sets of digitally encoded symbols. Arithmetic words consist of digit vectors (strings of digits).

Operator: Operator is arithmetic or logical operation that is performed on the operand given in instructions.

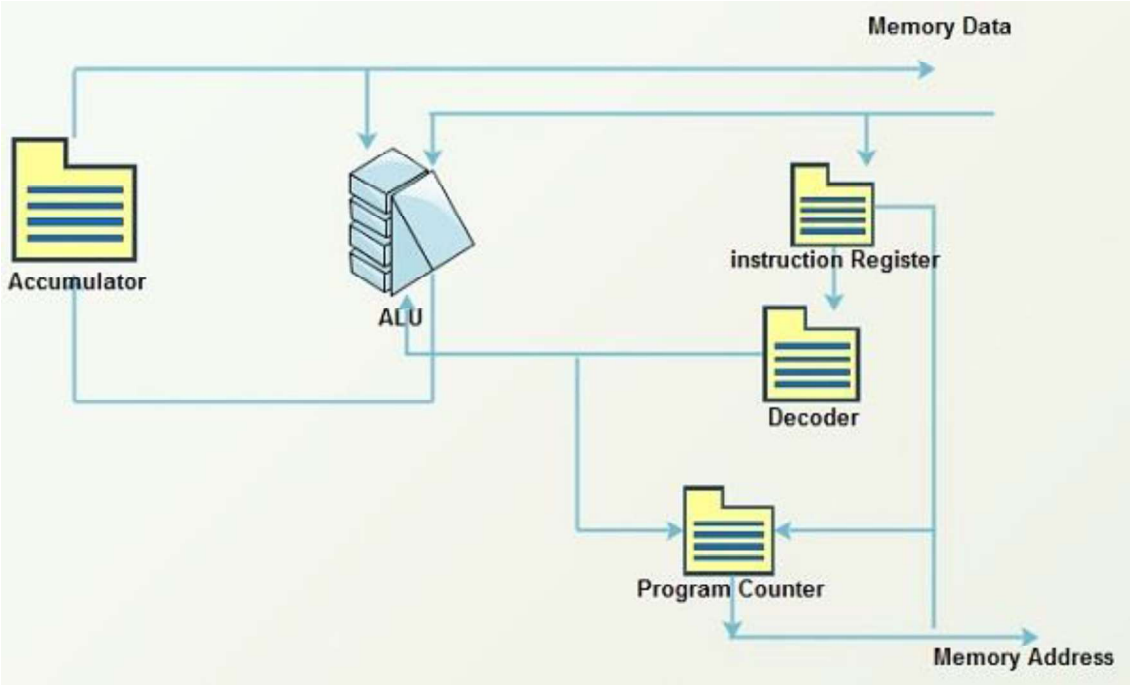
Flag: ALU uses many types of the flag during processing instructions. All these bits are stored in status or flag registers.

Arithmetic Logical Unit (ALU) Architecture

ALU is formed through the combinational circuit. The combinational circuit used logical gates like AND, OR, NOT, XOR for their construction. The combinational circuit does not have any memory element to store a previous data bit. Adders are the main part of the arithmetic logic unit to perform addition, subtraction by 2's complement.

Control unit generates the selection signals for selecting the function performed by ALU.

Registers : Registers are a very important component in ALU to store instruction, intermediate data, output, and input.



1.6 CONTROL UNIT

Definition: A **control unit** (**CU**) (or controller, same thing) is a piece of hardware that manages the activities of peripherals (separate devices attached to the computer, such as monitors, hard drives, printers, etc.) Control units found on personal computers are usually contained on a single printed circuit board. The control unit acts as a sort of "go-between," executing transfers of information between the computer's memory and the peripheral. Although the CPU (central processing unit-the "big boss" in the computer) gives instructions to the controller, it is the control unit itself that performs the actual physical transfer of data.

The control unit fetches one or more new instructions from memory (or an instruction cache), decodes them and dispatches them to the appropriate functional units to be executed. The

control unit is also responsible for setting the latches in various data paths that ensure that the instructions are performed on the correct operand values stored in the registers.

In a CISCprocessor, the control unit is a small processor in its own right that executes microcode programs stored in a region of rom that prescribe the correct sequence of latches and data transfers for each type of macroinstruction. A RISC processor does away with microcode and most of the complexity in the control unit, which is left with little more to do than decode the instructions and turn on the appropriate functional units.

Functions of control Unit

• Regulate transfers of information between memory and I/O.

• Fetches and decodes instructions from micro programs.

• Responsible for correct instruction execution between a processor's many sub-units.

• Control unit converts received information into sequence of control signals, and transfer to computer processor.

1.7 SUMMARY

In this unit we have introduced the concept of von-neumann architecture. We also discussed Instruction set architecture. At the end of this unit reader are able to figure out the concept of arithmetic and logic unit and control unit.

1.8 KEYWORDS

ALU, CU, Register, Flag, operator and ISA

1.9 QUESTIONS

1. Define computer architecture.
2. Discuss instruction set architecture.
3. Explain von-neumann architecture.
4. Write a short note on control unit.
5. Briefly explain arithmetic and logic unit.

1.10 REFERENCES

- John L. Hennessy and David A. Patterson, Computer Architecture: A Quantitative Approach, MorganKaufmann.
- Kai Hwang, Advanced Computer Architecture: Parallelism, Scalability, Programmability, McGraw-Hill.
- M. J. Flynn, Computer Architecture: Pipelined and Parallel Processor Design, Narosa PublishingHouse.

UNIT 2: MEMORY AND I/O SYSTEMS

Structure:

2.0 OBJECTIVES

After going through this lesson you will be able to

- Elucidate memory and I/O devices.
- Discuss interfacing to the CPU
- Describe measuring and reporting performance
- Discuss CISC and RISC processors

2.1 INTRODUCTION

Input-Output Interface is used as an method which helps in transferring of information between the internal storage devices i.e. memory and the external peripheral device . A peripheral device is that which provide input and output for the computer, it is also called Input-Output devices. **Computer performance** is the amount of work accomplished by a computer system. The word performance in computer performance means "How well is the computer doing the work it is supposed to do?". It basically depends on response time, throughput and execution time of a computer system.

2.2 MEMORY AND I/O DEVICES

A memory is just like a human brain. It is used to store data and instructions. Computer memory is the storage space in the computer, where data is to be processed and instructions required for

processing are stored. The memory is divided into large number of small parts called cells. Each location or cell has a unique address, which varies from zero to memory size minus one. For example, if the computer has 64k words, then this memory unit has 64 * 1024 = 65536 memory locations. The address of these locations varies from 0 to 65535.

Memory is primarily of three types −

- Cache Memory
- Primary Memory/Main Memory
- Secondary Memory

Cache Memory

Cache memory is a very high speed semiconductor memory which can speed up the CPU. It acts as a buffer between the CPU and the main memory. It is used to hold those parts of data and program which are most frequently used by the CPU. The parts of data and programs are transferred from the disk to cache memory by the operating system, from where the CPU can access them.

Advantages

The advantages of cache memory are as follows −

- Cache memory is faster than main memory.
- It consumes less access time as compared to main memory.
- It stores the program that can be executed within a short period of time.
- It stores data for temporary use.

Disadvantages

The disadvantages of cache memory are as follows −

- Cache memory has limited capacity.
- It is very expensive.

Primary Memory (Main Memory)

Primary memory holds only those data and instructions on which the computer is currently working. It has a limited capacity and data is lost when power is switched off. It is generally made up of semiconductor device. These memories are not as fast as registers. The data and instruction required to be processed resides in the main memory. It is divided into two subcategories RAM and ROM.

Characteristics of Main Memory

- These are semiconductor memories.
- It is known as the main memory.
- Usually volatile memory.
- Data is lost in case power is switched off.
- It is the working memory of the computer.
- Faster than secondary memories.
- A computer cannot run without the primary memory.

Secondary Memory

This type of memory is also known as external memory or non-volatile. It is slower than the main memory. These are used for storing data/information permanently. CPU directly does not access these memories, instead they are accessed via input-output routines. The contents of secondary memories are first transferred to the main memory, and then the CPU can access it. For example, disk, CD-ROM, DVD, etc.

Characteristics of Secondary Memory

- These are magnetic and optical memories.
- It is known as the backup memory.
- It is a non-volatile memory.
- Data is permanently stored even if power is switched off.
- It is used for storage of data in a computer.
- Computer may run without the secondary memory.
- Slower than primary memories.

Input/Output Subsystem

The I/O subsystem of a computer provides an efficient mode of communication between the central system and the outside environment. It handles all the input-output operations of the computer system.

Peripheral Devices

Input or output devices that are connected to computer are called **peripheral devices**. These devices are designed to read information into or out of the memoryunit upon command from the CPU and are considered to be the part of computer system. These devices are also called **peripherals**.For example: *Keyboards*, *display units* and *printers* are common peripheral devices.

There are three types of peripherals:

Input peripherals : Allows user input, from the outside world to thecomputer. Example: Keyboard, Mouse etc.
Output peripherals: Allows information output, from the computer to theoutside world. Example: Printer, Monitor etc
Input-Output peripherals: Allows both input(from outised world to computer) as well as, output(from computer to the outside world). Example:Touch screen etc.

Interfaces: Interface is a shared boundary btween two separate components of the computer system which can be used to attach two or more components to the system for communication purposes.

There are two types of interface:

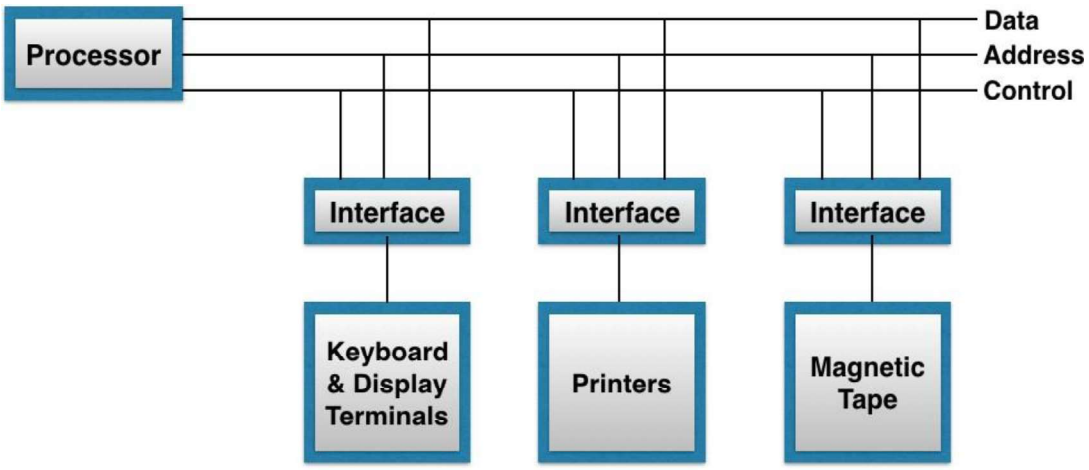
1. CPU Inteface
2. I/O Interface

2.3 INTERFACING TO THE CPU

The I/O interface supports a method by which data is transferred between internal storage and external I/O devices. All the peripherals connected to a computer require special communication connections for interfacing them with the CPU.**I/O Bus and Interface**

Modules. The I/O bus is the route used for peripheral devices to interact with the computer processor. A typical connection of the I/O bus to I/O devices is shown in the figure.

The I/O bus includes data lines, address lines, and control lines. In any general-purpose computer, the magnetic disk, printer, and keyboard, and display terminal are commonly employed. Each peripheral unit has an interface unit associated with it. Each interface decodes the control and address received from the I/O bus.



Connection of I/O Bus to I/O Device

It can describe the address and control received from the peripheral and supports signals for the peripheral controller. It also conducts the transfer of information between peripheral and processor and also integrates the data flow.

The I/O bus is linked to all peripheral interfaces from the processor. The processor locates a device address on the address line to interact with a specific device. Each interface contains an address decoder attached to the I/O bus that monitors the address lines.

When the address is recognized by the interface, it activates the direction between the bus lines and the device that it controls. The interface disables the peripherals whose address does not equivalent to the address in the bus.

An interface receives any of the following four commands −

- **Control** − A command control is given to activate the peripheral and to inform its next task. This control command depends on the peripheral, and each peripheral receives its sequence of control commands, depending on its mode of operation.

- **Status** − A status command can test multiple test conditions in the interface and the peripheral.
- **Data Output** − A data output command creates the interface counter to the command by sending data from the bus to one of its registers.
- **Data Input** − The data input command is opposite to the data output command. In data input, the interface gets an element of data from the peripheral and places it in its buffer register.

2.4 MEASURING AND REPORTING PERFORMANCE

Measuring and Reporting Performance: The computer user is interested in reducing response time(the time between the start and the completion of an event) also referred to as execution time. The manager of a large data processing center may be interested in increasing throughput(the total amount of work done in a given time).

Even execution time can be defined in different ways depending on what we count. The most straightforward definition of time is called wall-clock time, response time, or elapsed time, which is the latency to complete a task, including disk accesses, memory accesses, input/output activities, operating system overhead

Choosing Programs to Evaluate Performance

A computer user who runs the same programs day in and day out would be the perfect candidate to evaluate a new computer. To evaluate a new system the user would simply compare the execution time of her workload—the mixture of programs and operating system commands that users run on a machine.

There are five levels of programs used in such circumstances, listed below in decreasing order of accuracy of prediction.

1. Real applications— Although the buyer may not know what fraction of time is spent on these programs, she knows that some users will run them to solve real problems. Examples are compilers for C, text-processing software like Word, and other applications like Photoshop. Real applications have input, output, and options that a user can select when running the program. There is one major downside to using real applications as benchmarks: Real applications often encounter portability problems arising from dependences on the operating

system or compiler. Enhancing portability often means modifying the source and sometimes eliminating some important activity, such as interactive graphics, which tends to be more system-dependent.

2. Modified (or scripted) applications—In many cases, real applications are used as the building block for a benchmark either with modifications to the application or with a script that acts as stimulus to the application. Applications are modified for two primary reasons: to enhance portability or to focus on one particular aspect of system performance. For example, to create a CPU-oriented benchmark, I/O may be removed or restructured to minimize its impact on execution time. Scripts are used to reproduce interactive behavior, which might occur on a desktop system, or to simulate complex multiuser interaction, which occurs in a server system.

Kernels—Several attempts have been made to extract small, key pieces from real programs and use them to evaluate performance. Livermore Loops and Linpack are the best known examples. Unlike real programs, no user would run kernel programs, for they exist solely to evaluate performance. Kernels are best used to isolate performance of individual features of a machine to explain the reasons for differences in performance of real programs.

4. Toy benchmarks—Toy benchmarks are typically between 10 and 100 lines of code and produce a result the user already knows before running the toy program. Programs like Sieve of Eratosthenes, Puzzle, and Quicksort are popular because they are small, easy to type, and run on almost any computer. The best use of such programs is beginning programming assignments

5. Synthetic benchmarks—Similar in philosophy to kernels, synthetic benchmarks try to match the average frequency of operations and operands of a large set of programs. Whetstone and Dhrystone are the most popular synthetic benchmarks.

Benchmark Suites

Recently, it has become popular to put together collections of benchmarks to try to measure the performance of processors with a variety of applications. One of the most successful attempts to create standardized benchmark application suites has been the SPEC (Standard Performance Evaluation Corporation), which had its roots in the late 1980s efforts to deliver

better benchmarks for workstations. Just as the computer industry has evolved over time, so has the need for different benchmark suites, and there are now SPEC benchmarks to cover different application classes, as well as other suites based on the SPEC model. Which is shown in figure

Desktop Benchmarks: Desktop benchmarks divide into two broad classes: CPU intensive benchmarks and graphics intensive benchmarks intensive CPU activity). SPEC originally created a benchmark set focusing on CPU performance (initially called SPEC89), which has evolved into its fourth generation: SPEC CPU2000, which follows SPEC95, and SPEC92.

Although SPEC CPU2000 is aimed at CPU performance, two different types of graphics benchmarks were created by SPEC: SPEC viewperf is used for benchmarking systems supporting the OpenGL graphics library, while SPECapc consists of applications that make extensive use of graphics. SPECviewperf measures the 3D rendering performance of systems running under OpenGL using a 3-D model and a series of OpenGL calls that transform the model. SPECapc consists of runs of three large applications:

1. Pro/Engineer: a solid modeling application that does extensive 3-D rendering. The input script is a model of a photocopying machine consisting of 370,000 triangles.
2. SolidWorks 99: a 3-D CAD/CAM design tool running a series of five tests varying from I/O intensive to CPU intensive. The largetest input is a model of an assembly line consisting of 276,000 triangles.
3. Unigraphics V15: The benchmark is based on an aircraft model and covers a wide spectrum of Unigraphics functionality, including assembly, drafting, numeric control machining, solid modeling, and optimization. The inputs are all part of an aircraft design.

Server Benchmarks: Just as servers have multiple functions, so there are multiple types of benchmarks. The simplest benchmark is perhaps a CPU throughput oriented benchmark. SPEC CPU2000 uses the SPEC CPU benchmarks to construct a simple throughput benchmark where the processing rate of a multiprocessor can be measured by running multiple copies (usually as many as there are CPUs) of each SPEC CPU benchmark and converting the CPU time into a rate. This leads to a measurement called the SPECRate. Other than SPECRate, most server applications and benchmarks have significant I/O activity arising from either disk or network traffic, including benchmarks for file server systems, for web servers, and for database and transaction processing systems. SPEC offers both a file server benchmark (SPECSFS) and a

web server benchmark (SPECWeb). SPECSFS (see http://www.spec.org/osg/sfs93/) is a benchmark for measuring NFS (Network File System) performance using a script of file server requests; it tests the performance of the I/O system (both disk and network I/O) as well as the CPU. SPECSFS is a throughput oriented benchmark but with important response time requirements.

Transaction processing benchmarks measure the ability of a system to handle transactions, which consist of database accesses and updates. All the TPC benchmarks measure performance in transactions per second. In addition, they include a response-time requirement, so that throughput performance is measured only when the response time limit is met. To model real-world systems, higher transaction rates are also associated with larger systems, both in terms of users and the data base that the transactions are applied to. Finally, the system cost for a benchmark system must also be included, allowing accurate comparisons of cost-performance.

Embedded Benchmarks

Benchmarks for embedded computing systems are in a far more nascent state than those for either desktop or server environments. In fact, many manufacturers quote Dhrystone performance, a benchmark that was criticized and given up by desktop systems more than 10 years ago! As mentioned earlier, the enormous variety in embedded applications, as well as differences in performance requirements (hard real-time, soft real-time, and overall cost-performance), make the use of a single set of benchmarks unrealistic.

In practice, many designers of embedded systems devise benchmarks that reflect their application, either as kernels or as stand-alone versions of the entire application. For those embedded applications that can be characterized well by kernel performance, the best standardized set of benchmarks appears to be a new benchmark set: the EDN Embedded Microprocessor Benchmark Consortium (or EEMBC–pronounced embassy). The EEMBC benchmarks fall into five classes: automotive/industrial, consumer, networking, office automation, and telecommunications Figure shows the five different application classes, which include 34 benchmarks.

2.5 CISC AND RISC PROCESSORS

CISC Approach

The primary goal of CISC architecture is to complete a task in as few lines of assembly as possible. This is achieved by building processor hardware that is capable of understanding and executing a series of operations. For this particular task, a CISC processor would come prepared with a specific instruction (we'll call it "MULT"). When executed, this instruction loads the two values into separate registers, multiplies the operands in the execution unit, and then stores the product in the appropriate register. Thus, the entire task of multiplying two numbers can be completed with one instruction:

MULT 2:3, 5:2

MULT is what is known as a "complex instruction." It operates directly on the computer's memory banks and does not require the programmer to explicitly call any loading or storing functions. It closely resembles a command in a higher level language. For instance, if we let "a" represent the value of 2:3 and "b" represent the value of 5:2, then this command is identical to the C statement "a = a * b."

One of the primary advantages of this system is that the compiler has to do very little work to translate a high-level language statement into assembly. Because the length of the code is relatively short, very little RAM is required to store instructions. The emphasis is put on building complex instructions directly into the hardware.

RISC Approach

RISC processors only use simple instructions that can be executed within one clock cycle. Thus, the "MULT" command described above could be divided into three separate commands: "LOAD," which moves data from the memory bank to a register, "PROD," which finds the product of two operands located within the registers, and "STORE," which moves data from a register to the memory banks. In order to perform the exact series of steps described in the CISC approach, a programmer would need to code four lines of assembly:

LOAD A, 2:3
LOAD B, 5:2
PROD A, B
STORE 2:3, A

At first, this may seem like a much less efficient way of completing the operation. Because there are more lines of code, more RAM is needed to store the assembly level instructions. The compiler must also perform more work to convert a high-level language statement into code of this form.

However, the RISC strategy also brings some very important advantages. Because each instruction requires only one clock cycle to execute, the entire program will execute in approximately the same amount of time as the multi-cycle "MULT" command. These RISC "reduced instructions" require less transistors of hardware space than the complex instructions, leaving more room for general purpose registers. Because all of the instructions execute in a uniform amount of time (i.e. one clock), pipelining is possible.

Separating the "LOAD" and "STORE" instructions actually reduces the amount of work that the computer must perform. After a CISC-style "MULT" command is executed, the processor automatically erases the registers. If one of the operands needs to be used for another computation, the processor must re-load the data from the memory bank into a register. In RISC, the operand will remain in the register until another value is loaded in its place.

The Performance Equation

The following equation is commonly used for expressing a computer's performance ability:

$$\frac{\text{time}}{\text{program}} = \frac{\text{time}}{\text{cycle}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{instructions}}{\text{program}}$$

The CISC approach attempts to minimize the number of instructions per program, sacrificing the number of cycles per instruction. RISC does the opposite, reducing the cycles per instruction at the cost of the number of instructions per program.

2.6 SUMMARY

At the end of this unit we have learnt Memory and I/O devices; we dealt with the concept on Interfacing to the CPU. In the later stage we focused measuring and reporting performance. In the last section of this unit we have discussed CISC and RISC architecture.

2.7 KEYWORDS

CISC, RISC, Cache memory, LOAD, input and output

2.8 QUESTIONS

1. Briefly explain memory and I/O devices.
2. With neat diagram explain interfacing to the CPU.
3. Explain server benchmarks.
4. Differentiate RISC and CISC approaches.

2.9 REFERENCES

- John L. Hennessy and David A. Patterson, Computer Architecture: A Quantitative Approach, MorganKaufmann.
- Kai Hwang, Advanced Computer Architecture: Parallelism, Scalability, Programmability, McGraw-Hill.
- M. J. Flynn, Computer Architecture: Pipelined and Parallel Processor Design, Narosa PublishingHouse.

UNIT 3: PIPELINING AND HAZARDS

Structure:

3.0 OBJECTIVES

After going through this lesson you will be able to

- Describe basic concepts of pipelining
- Discuss data hazard, control hazard and structural hazard.
- Elucidate Techniques for overcoming or reducing the effects of various hazards

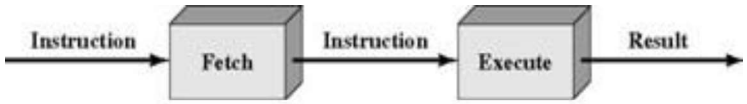
3.1 INTRODUCTION

Pipelining defines the temporal overlapping of processing. Pipelines are emptiness greater than assembly lines in computing that can be used either for instruction processing or, in a more general method, for executing any complex operations. It can be used efficiently only for a sequence of the same task, much similar to assembly lines. Hazards are problems with the instruction pipeline in CPU micro architectures when the next instruction cannot execute in the following clock cycle, and can potentially lead to incorrect computation results. Three common types of hazards are data hazards, structural hazards, and control hazards (branching hazards).

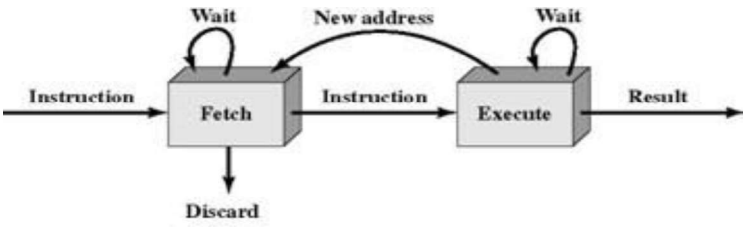
3.2 BASIC CONCEPTS OF PIPELINING

INSTRUCTION PIPELINING

As computer systems evolve, greater performance can be achieved by taking advantage of improvements in technology, such as faster circuitry, use of multiple registers rather than a single accumulator, and the use of a cache memory. Another organizational approach is instruction pipelining in which new inputs are accepted at one end before previously accepted inputs appear as outputs at the other end.



3.1(a) Simplified View



3.1(b) Expanded View

3.1 Two-Stage Instruction Pipeline

Figure 3.1a depicts this approach. The pipeline has two independent stages. The first stage fetches aninstruction and buffers it. When the second stage is free, the first stage passes it the buffered instruction. While the second stage is executing the instruction, the first stage takes advantage of any unused memory cycles to fetch and buffer the next instruction. This is called

instruction prefetch or *fetch overlap.*

This process will speed up instruction execution only if the fetch and execute stages were of equal duration, the instruction cycle time would be halved. However, if we look more closely at this pipeline (Figure 3.1b), we will see that this doubling of execution rate is unlikely for 3 reasons:

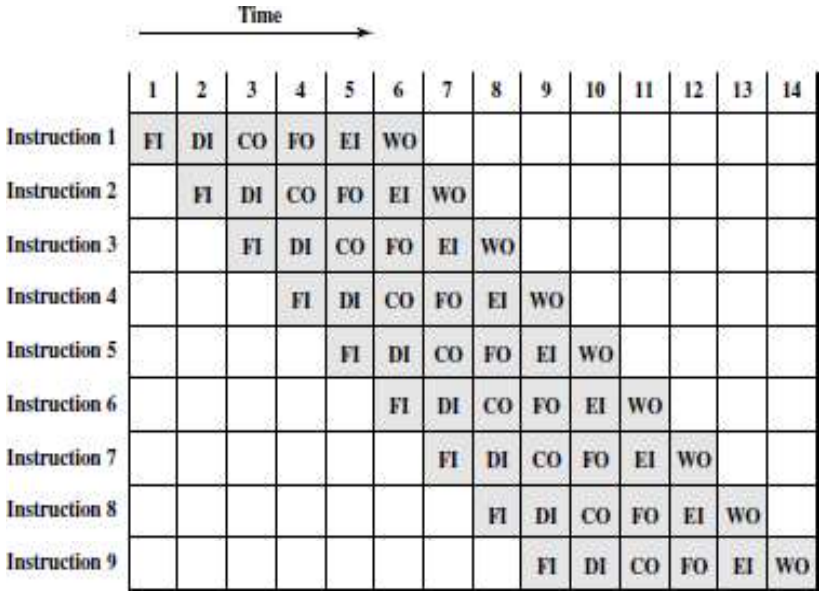
1. The execution time will generally be longer than the fetch time. Thus, the fetch stage may haveto wait for some time before it can empty its buffer.
2. A conditional branch instruction makes the address of the next instruction to be fetched unknown. Thus, the fetch stage must wait until it receives the next instruction address from the execute stage. The execute stage may then have to wait while the next instruction is fetched.

When a conditional branch instruction is passed on from the fetch to the execute stage, the fetch stage fetches the next instruction in memory after the branch instruction. Then, if the branch is not taken, no time is lost .If the branch is taken, the fetched instruction must be discarded and a new instruction fetched.

To gain further speedup, the pipeline must have more stages. Let us consider the following decomposition of the instruction processing.

1. **Fetch instruction (FI):** Read the next expected instruction into a buffer.
2. **Decode instruction (DI):** Determine the opcode and the operand specifiers.
3. **Calculate operands (CO):** Calculate the effective address of each source operand. This mayinvolve displacement, register indirect, indirect, or other forms of address calculation.
4. **Fetch operands (FO):** Fetch each operand from memory.
5. **Execute instruction (EI):** Perform the indicated operation and store the result, if any, in thespecified destination operand location.
6. **Write operand (WO):** Store the result in memory.

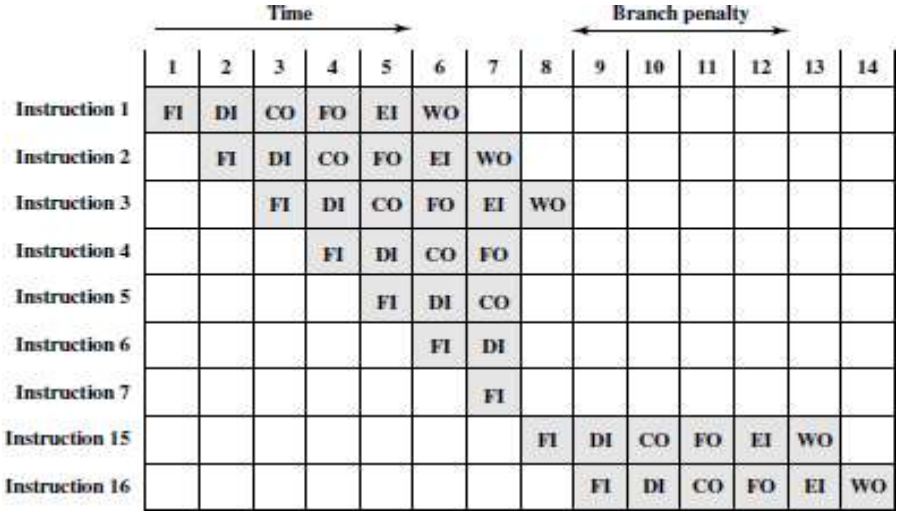
Figure 3.2 shows that a six-stage pipeline can reduce the execution time for 9 instructions



from 54 timeunits to 14 time units.

Timing Diagram for Instruction Pipeline Operation

FO and WO stages involve a memory access. If the six stages are not of equal duration, there will be some waiting involved at various pipeline stages. Another difficulty is the conditional



branch instruction, which can invalidate several instruction fetches. A similar unpredictable event is an interrupt.

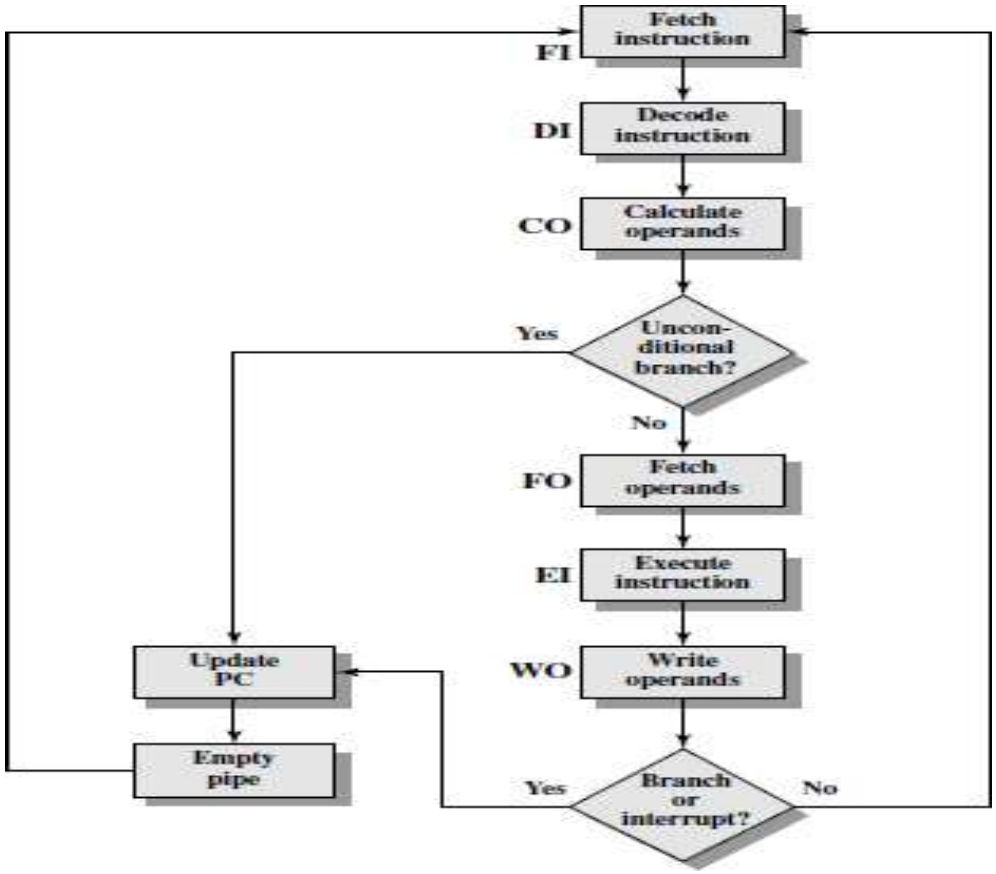
Timing Diagram for Instruction Pipeline Operation with interrupts

Figure 3.3 illustrates the effects of the conditional branch, using the same program as Figure 3.2. Assume that instruction 3 is a conditional branch to instruction 15. Until the instruction is

executed, there is no way of knowing which instruction will come next. The pipeline, in this example, simply loads the next instruction in sequence (instruction 4) and proceeds.

In Figure 3.2, the branch is not taken. In Figure 3.3, the branch is taken. This is not determined until the end of time unit 7.At this point, the pipeline must be cleared of instructions that are not useful. Duringtime unit 8, instruction 15 enters the pipeline.

No instructions complete during time units 9 through 12; this is the performance penalty incurred because we could not anticipate the branch. Figure 3.4 indicates the logic needed for



pipelining to accountfor branches and interrupts.

Six-stage CPU Instruction Pipeline

Figure 3.5 shows same sequence of events, with time progressing vertically down the figure, and each row showing the state of the pipeline at a given point in time. In Figure 3.5a (which corresponds to Figure 3.2), the pipeline is full at time 6, with 6 different instructions in various stages of execution, and remains full through time 9; we assume that instruction I9 is the last instruction to be executed. In Figure 3.5b, (which corresponds to Figure 3.3), the pipeline is

full at times 6 and 7. At time 7, instruction 3 is in the execute stage and executes a branch to instruction 15. At this point, instructions I4 through I7 are flushed from the pipeline, so that at time 8, only two instructions are in the pipeline, I3 and I15.

For high-performance in pipelining designer must still consider about :

1. At each stage of the pipeline, there is some overhead involved in moving data from buffer to buffer and in performing various preparation and delivery functions. This overhead can appreciably lengthen the total execution time of a single instruction.
2. The amount of control logic required to handle memory and register dependencies and tooptimize the use of the pipeline increases enormously with the number of stages. This can lead to a situation where the logic controlling the gating between stages is more complex than the stages being controlled.
3. Latching delay: It takes time for pipeline buffers to operate and this adds to instruction cycle time.

3.3 DATA HAZARDS

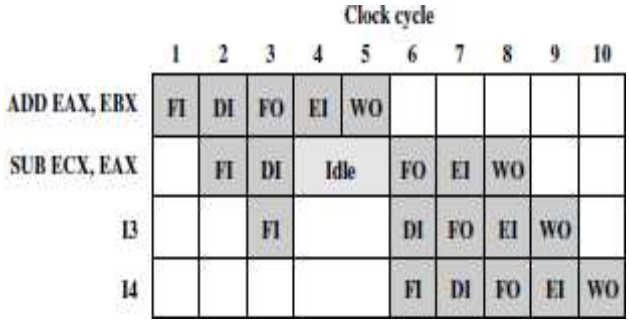
DATA HAZARDS A data hazard occurs when two instructions in a program are to be executed in sequence and both access a particular memory or register operand. If the two instructions are executed in strict sequence, no problem occurs but if the instructions are executed in a pipeline, then the operandvalue is to be updated in such a way as to produce a different result than would occur only with strict sequential execution of instructions. The program produces an incorrect result because of the use of pipelining.

As an example, consider the following x86 machine instruction sequence:

ADD EAX, EBX /* EAX = EAX + EBXSUB ECX, EAX /* ECX = ECX - EAX

The first instruction adds the contents of the 32-bit registers EAX and EBX and stores the result in EAX. The second instruction subtracts the contents of EAX from ECX and stores the result in ECX.

Figure 3.7 shows the pipeline behaviour. The ADD instruction does not update register EAX until the end of stage 5, which occurs at clock cycle 5. But the SUB instruction needs that value at the beginning of its stage 2, which occurs at clock cycle 4. To maintain correct operation, the pipeline must stall for two clocks cycles. Thus, in the absence of special hardware and specific avoidance



algorithms, such a data hazard results in inefficient pipeline usage. There are three types of data hazards;

Example of Resource Hazard

1. **Read after write (RAW), or true dependency:**.A hazard occurs if the read takes place before thewrite operation is complete.
2. **Write after read (RAW), or antidependency:** A hazard occurs if the write operation completesbefore the read operation takes place.
3. **Write after write (RAW), or output dependency:** Two instructions both write to the samelocation. A hazard occurs if the write operations take place in the reverse order of the intended sequence. The example of Figure 3.7 is a RAW hazard.

3.4 CONTROL HAZARDS

CONTROL HAZARDS A control hazard, also known as a *branch hazard,* occurs when the pipeline makes the wrong decision on a branch prediction and therefore brings instructions into the pipeline that must subsequently be discarded.

Dealing with Branches

Until the instruction is actually executed, it is impossible to determine whether the branch will be taken ornot. A variety of approaches have been taken for dealing with conditional branches:

- Multiple streams
- Prefetch branch target
- Loop buffer
- Branch prediction
- Delayed branch

MULTIPLE STREAMS A simple pipeline suffers a penalty for a branch instruction because it mustchoose one of two instructions to fetch next and may make the wrong choice.

A brute-force approach is to replicate the initial portions of the pipeline and allow the pipeline to fetch both instructions, making use of two streams. There are two problems with this approach:

With multiple pipelines there are contention delays for access to the registers and to memory. Additional branch instructions may enter the pipeline (either stream) before the original branch decision is resolved. Each such instruction needs an additional stream.

Examples of machines with two or more pipeline streams are the IBM 370/168 and the IBM 3033.

PREFETCH BRANCH TARGET When a conditional branch is recognized, the target of the branch is prefetched, in addition to the instruction following the branch. This target is then saved until the branch instruction is executed. If the branch is taken, the target has already been prefetched.

Example- The IBM 360/91 uses this approach.

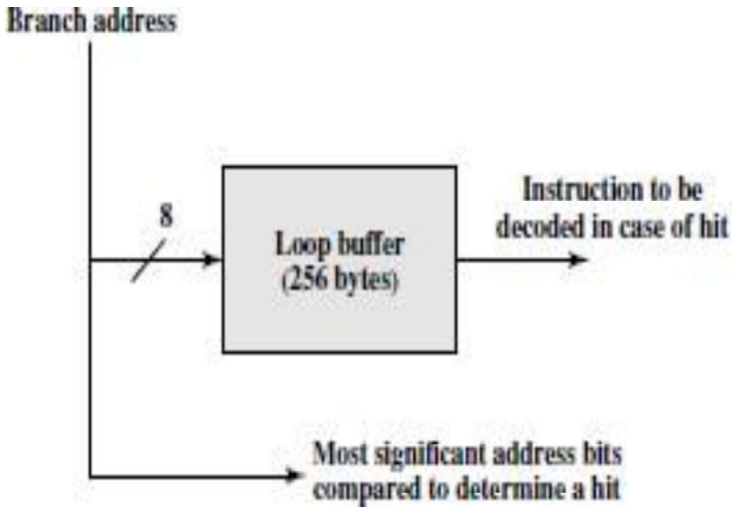
LOOP BUFFER A loop buffer is a small, very-high-speed memory maintained by the instruction fetch stage of the pipeline and containing the n most recently fetched instructions, in sequence. If a branch is to be taken, the hardware first checks whether the branch target is within the buffer. If so, the nextinstruction is fetched from the buffer.

The loop buffer has three benefits:

1. With the use of prefetching, the loop buffer will contain some instruction sequentially ahead of the current instruction fetch address.

2. If a branch occurs to a target just a few locations ahead of the address of the branch instruction,the target will already be in the buffer.
3. This strategy is particularly well suited to dealing with loops, or iterations; hence the name *loop buffer*. If the loop buffer is large enough to contain all the instructions in a loop, then those instructions need to be fetched from memory only once, for the first iteration. For subsequent iterations, all the needed instructions are already in the buffer.

Figure 3.8 gives an example of a loop buffer



Loop Buffer

BRANCH PREDICTION Various techniques can be used to predict whether a branch will be taken. Among the more common are the following:

- Predict never taken
- Predict always taken
- Predict by opcode
- Taken/not taken switch
- Branch history table

The first three approaches are static: they do not depend on the execution history up to the time of the conditional branch instruction. The latter two approaches are dynamic: They depend on the execution history.

The first two approaches are the simplest. These either always assume that the branch will not be taken or continue to fetch instructions in sequence, or they always assume that the branch will be taken and always fetch from the branch target. The predict-never-taken approach is the most popular of all the branch prediction methods.

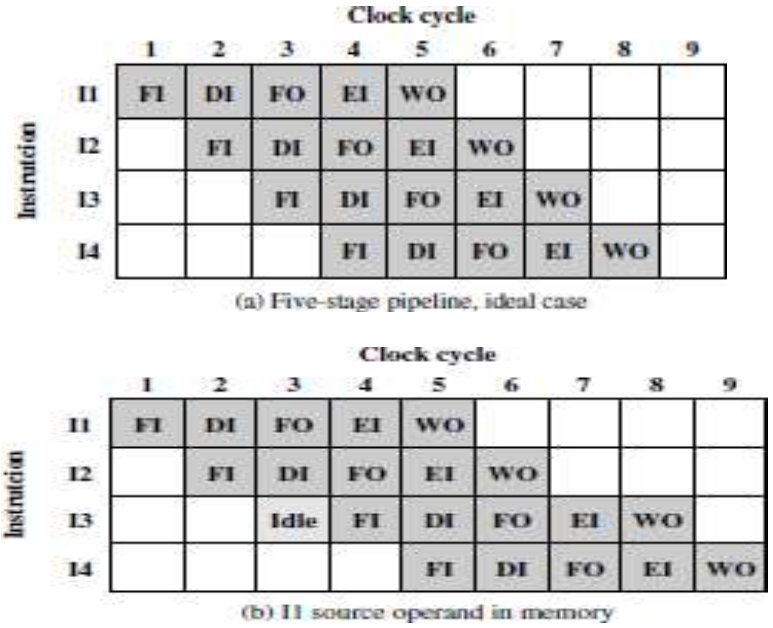
DELAYED BRANCH It is possible to improve pipeline performance by automatically rearranging instructions within a program, so that branch instructions occur later than actually desired.

3.5 STRUCTURAL HAZARDS

A **pipeline hazard** occurs when the pipeline, or some portion of the pipeline, must stall because conditions do not permit continued execution. Such a pipeline stall is also referred to as a *pipeline bubble*.There are three types of hazards: resource, data, and control.

RESOURCE HAZARDS A resource hazard occurs when two (or more) instructions that are already in the pipeline need the same resource. The result is that the instructions must be executed in serial rather than parallel for a portion of the pipeline. A resource hazard is sometime referred to as a *structural hazard*.

Let us consider a simple example of a resource hazard.Assume a simplified five-stage pipeline, in which each stage takes one clock cycle. In Figure 3.6a which a new instruction enters the pipeline each clock cycle. Now assume that main memory has a single port and that all instruction fetches and data reads and writes must be performed one at a time. In this case, an operand read to or write from memory cannot be performed in parallel with an instruction fetch. This is illustrated in Figure 3.6b, which assumesthat the source operand for instruction I1 is in memory, rather than a register. Therefore, the fetch in- struction stage of the pipeline must idle for one cycle before beginning the instruction fetch for instructionI3. The figure assumes that all other operands are in registers.



Example of Resource Hazard

3.6 TECHNIQUES FOR OVERCOMING OR REDUCING THE EFFECTS OF VARIOUS HAZARDS

Data Hazards occur when an instruction depends on the result of previous instruction and that result of instruction has not yet been computed. whenever two different instructions use the same storage. the location must appear as if it is executed in sequential order.

There are four types of data dependencies: Read after Write (RAW), Write after Read (WAR), Write after Write (WAW), and Read after Read (RAR). These are explained as follows below.

Read after Write (RAW) :

It is also known as True dependency or Flow dependency. It occurs when the value produced by an instruction is required by a subsequent instruction. For example,

ADD R1, --, --;

SUB --, R1, --;

Stalls are required to handle these hazards.

Write after Read (WAR) :

It is also known as anti dependency. These hazards occur when the output register of an instruction is used right after read by a previous instruction. For example,

ADD --, R1, --;

SUB R1, --, --;

Write after Write (WAW) :

It is also known as output dependency. These hazards occur when the output register of an instruction is used for write after written by previous instruction. For example,

ADD R1, --, --;

SUB R1, --, --;

Read after Read (RAR) :

It occurs when the instruction both read from the same register. For example,

ADD --, R1, --;

SUB --, R1, --;

Since reading a register value does not change the register value, these Read after Read (RAR) hazards don't cause a problem for the processor.

Handling Data Hazards :

These are various methods we use to handle hazards: Forwarding, Code recording, and Stall insertion.

These are explained as follows below.

1. **Forwarding:** It adds special circuitry to the pipeline. This method works because it takes less time for the required values to travel through a wire than it does for a pipeline segment to compute its result.
2. **Code reordering:** We need a special type of software to reorder code. We call this type of software a hardware-dependent compiler.

3. **Stall Insertion:** it inserts one or more installs (no-op instructions) into the pipeline, which delays the execution of the current instruction until the required operand is written to the register file, but this method decreases pipeline efficiency and throughput.

3.7 SUMMARY

In this unit we have discussed in detail about basics of pipelining and also covered data hazard, control hazard and structural hazard. At the end of this unit also covered in detail techniques for overcoming or reducing the effects of various hazards.

3.8 KEYWORDS

Data hazard, Control hazard, Branch hazard and loop buffer

3.9 QUESTIONS

1. Discuss Pipelining.
2. Describe data hazard.
3. Elucidate control hazard.
4. Briefly explain techniques for overcoming or reducing the effects of various hazards.

3.10 REFERENCES

- John L. Hennessy and David A. Patterson, Computer Architecture: A Quantitative Approach, MorganKaufmann.
- Kai Hwang, Advanced Computer Architecture: Parallelism, Scalability, Programmability, McGraw-Hill.
- M. J. Flynn, Computer Architecture: Pipelined and Parallel Processor Design, Narosa PublishingHouse.

UNIT 4: MEMORY LOCATIONS, ADDRESSES AND OPERATIONS

Structure

4.0 OBJECTIVES

At the end of this unit, you should be able to

- Define word, word length, memory address
- Differentiate between Big-endian and little-endian assignments
- Explain the way of accessing Numbers, Characters and Character strings from the main memory
- Cleary indicate as to what constitutes a memory operation

4.1 INTRODUCTION

So far in previous units, we saw the general concepts related to computer organization and the way a computer is made operational. In this unit, we shall see how memory locations are organized and addressed. More specifically, we shall look into the two basic types of assignments one, the big-endian and the other, the little-endian. Memory operations play a very important role in the process of executing a program or operating on the data. The two important operations, Load and Store are explained with reference to the memory.

4.2 MEMORY LOCATIONS AND ADDRESSES

The main memory consists of a large number of storage cells, each of which can store a binary digit 0 or 1. This 1-bit representation of information is too small to be handled by a computer. So, a group of n bits are used while storing or retrieving. Each group of n bits is referred to as its word length. The memory of a computer is schematically represented as a collection of words as shown in figure 4.1.

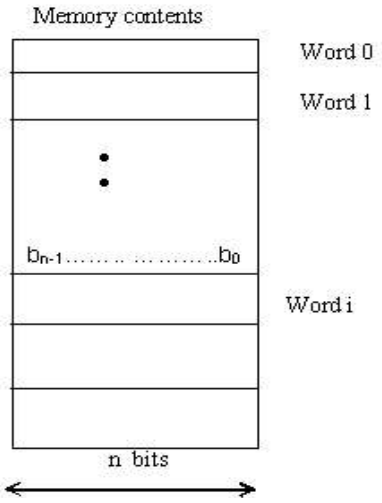
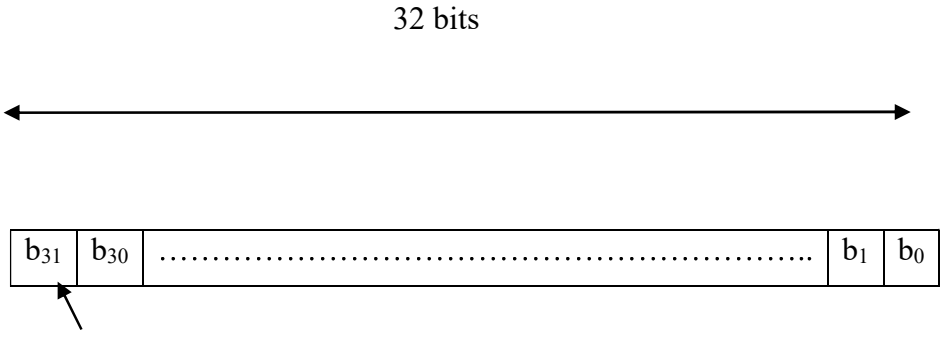


Figure 4.1 Main memory words

The word length of modern computers can typically have 8 bits, 16 bits, 32, 64 or 128 bits depending on the size of the computer. If the word length is 32 bits, a single word can store a 32-bit 2's complement number or four ASCII characters each with 8 bits. The method of encoding information in 32-bit word is shown in figure 4.2(a) and figure 4.2(b).



Sign bit: $b_{31} = 0$ for positive numbers

b_{31} = 1 for negative numbers

Figure 4.2(a) Encoding a signed integer information in a 32-bit word

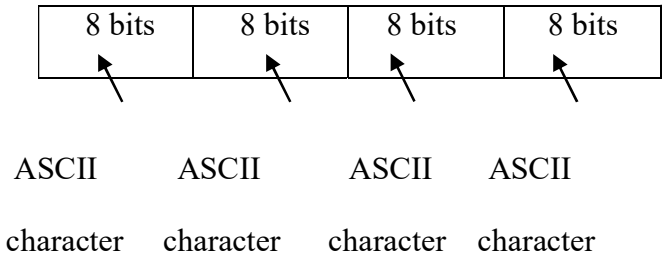


Figure 4.2(b) Encoding character information in a 32-bit word.

Each location in the main memory is given a distinct name or address. With this address it is possible to store or retrieve information from the memory location. The contents of memory locations can represent either instructions or operands. The operands can be numbers or characters. Numbers from 0 through 2^k -1, with some value for k can be used as addresses of successive locations in memory. The 2^k addresses constitute the address space of computer where in we can access memory up to 2^k addressable locations. The table 4.1 gives information on address space and addressable location for n-bit address.

Table 4.1 Address space and addressable location for n-bit address.

Bit address	**Address Space**	**Locations**	**Term**
10-bit	2^{10}	1024	1 Kilo

20-bit	2^{20}	1048576	1 Mega
30-bit	2^{30}	1073741824	1 Giga
40-bit	2^{40}	1099511627776	1 Tera

BYTE addressability:

We are now familiar with three information quantities such as bit, byte and word. In general, while designing instructions for a processor, we typically consider word length in the range of 16 to 64 bits. It is highly not possible for us to assign distinct addresses to individual bit locations in the memory. In the modern computers, the most preferred way of assigning distinct addresses is to have successive addresses refer to successive byte locations in the memory. This type of address assignment is known as byte-addressable memory. Byte locations have addresses 0, 1, 2,…… Hence, if the word length of a computer is 32 bits, the successive words are located at addresses 0,4, 8, 12, ….., with each word consisting of four bytes.

Big-Endian and Little-Endian Assignments:

The byte addressable memory assignment can be done in two ways. They are: big-endian assignment and little-endian assignment. In both the assignments, byte addresses 0, 4, 8, 12 …… are taken as the addresses of the successive words in memory and these addresses are used while specifying memory read and write operations.

Big-endian assignment: when the lower byte addresses are used for representing more significant bytes (the leftmost bytes) of the word, it is refereed to as big-endian notation. This type of assignment is used in commercial machines. This concept is illustrated in figure 4.3.

Word address Byte address

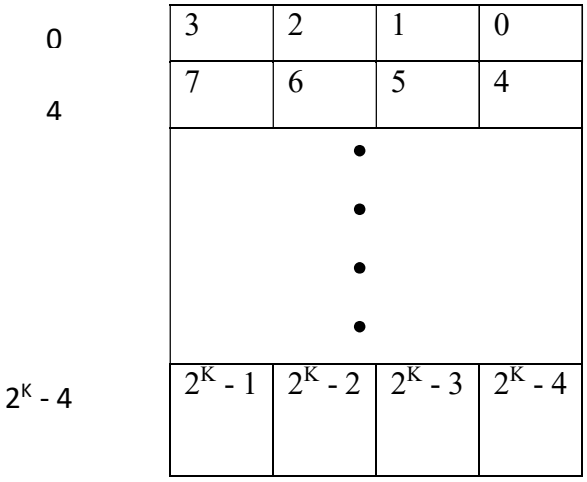


Figure 4.3 Big-endian assignments

Little-endian assignment: when the lower byte addresses are used for less significant bytes (the rightmost bytes) of the word, it is refereed to as little-endian notation. This type of assignment is also used in commercial machines. This concept is illustrated in figure 4.4.

Word address Byte address

0

4

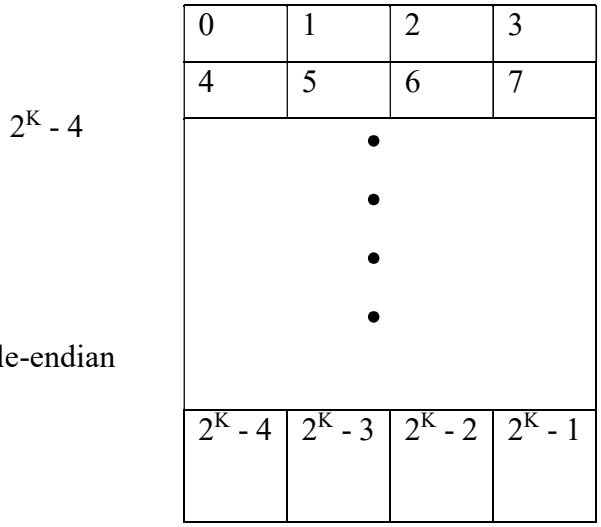


Figure 4.4 Little-endian assignments

Word alignment:

We have seen in the discussion of byte addressable memory assignments that in the case of a 32-bit word length instruction, the word boundaries occur at addresses 0, 4, 8, …. etc as shown in figure 4.3 and figure 4.4. We say that the word locations have aligned addresses. In general, words are said to be aligned in memory if they begin at a byte address that is a multiple of the number of bytes contained in a word.

For example, if the word length is 16 bits (or 2^2 bytes) then aligned words begin at byte addresses 0, 2, 4,…… And if the word length is 64 bits (or 2^3 bytes) then aligned words begin at byte addresses 0, 8, 16,…..etc.

Accessing Numbers, Characters & Character strings:

We know that, each type of data occupies differently in the memory and hence can be accessed differently. A numeric data usually occupies one word. It can be accessed in the memory by specifying its word address. Similarly, a character is stored in the memory as a byte and hence it