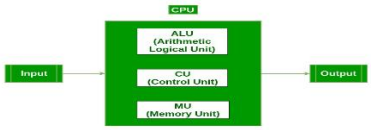
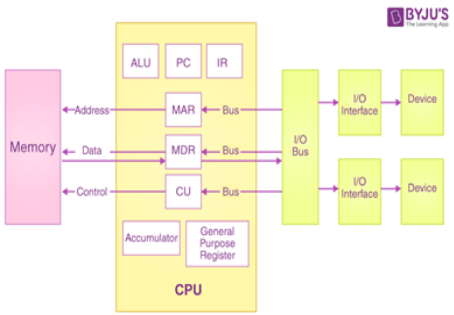
A computer has five functional units:
Input unit: Consists of input devices that convert data into binary language. Input devices include keyboards, mice, joysticks, and scanners.
Memory unit: Stores program information.
Arithmetic and logic unit: Also known as the ALU.
Output unit: The fifth functional unit.
Control unit: A functionally independent main part.



❑**Von Neumann architecture:**
• Von Neumann architecture was first published by John von Neumann in 1945. • His computer architecture design consists of a Control Unit, Arithmetic and Logic Unit (ALU), Memory Unit, Registers and Inputs/Outputs. • Historically there have been 2 types of Computers: 1. Fixed Program Computers – Their function is very specific and they couldn't be reprogrammed, e.g. Calculators. 2. Stored Program Computers – These can be programmed to carry out many different tasks, applications are stored on them, hence the name. • Von Neumann architecture is based on the stored-program computer concept, where instruction data and program data are stored in the same memory. This design is still used in most computers produced today.



Basic CPU Structure of ALU

➢ **Central Processing Unit (CPU):** ▫ The Central Processing Unit (CPU) is the electronic circuit responsible for executing the instructions of a computer program. ▫ It is sometimes referred to as the microprocessor or processor. ▫ The CPU contains the ALU, CU and a variety of registers.
▪ **Registers:** Registers are high speed storage areas in the CPU. All data must be stored in a register before it can be processed. MAR Memory Address Register Holds the memory location of data that needs to be accessed MDR Memory Data Register Holds data that is being transferred to or from memory AC Accumulator Where intermediate arithmetic and logic results are stored PC Program Counter Contains the address of the next instruction to be executed CIR Curren Instruction Register Contains the current instruction during processing
▪ **Arithmetic and Logic Unit (ALU):** The ALU allows arithmetic (add, subtract etc) and logic (AND, OR, NOT etc) operations to be carried out.
▪ Control Unit (CU): The control unit controls the operation of the computer's ALU, memory and input/output devices, telling them how to respond to the program instructions it has just read and interpreted from the memory unit. The control unit also provides the timing and control signals required by other computer components. ➢ **Memory Unit:** ▫ The memory unit is usually primary memory. Inside the primary memory consists of the Random Access Memory (RAM) and the Read-Only Memory (ROM).
▫ **The RAM** is used to store data that is currently in use. This is when the computer is on, data that is used is added into the RAM. However, since this is a volatile (temporary) memory, once the computer is off, all the data that was in the RAM is lost.
▫ **The ROM** is used to store permanent data and basic instructions such as the BIOS/startup instructions for your computer. This is different to RAM as the memory is non-volatile (permanent), therefore even when the computer is off, all these data is stored and retained in the ROM.
➢ **Input/Output Devices:** ▫ Input Devices — devices that sends information into the computer, eg: keyboard, mouse, microphone, touchscreen, etc. ▫ Output devices — devices that sends information out of the computer, eg: monitor, speaker, printer, etc.
Buses
υ Data is transmitted from one part of a computer to another, connecting all major internal components to the CPU and memory, by the means of Buses. υ A standard CPU system bus is comprised of a control bus, data bus and address bus.
Types of Buses
Address Bus Carries the addresses of data (but not the data) between the processor and memory
Data Bus Carries data between the processor, the memory unit and the Input/Output devices
Control Bus Carries control signals/commands from the CPU (and status signals from other devices) in order to control and coordinate all the activities within the computer.
von Neumann Bottleneck: Limitations of von Neumann Architecture υ It is the computing system throughput limitation due to inadequate rate of data transfer between memory and the CPU.
υ The VNB causes CPU to wait and idle for a certain amount of time while low speed memory is being accessed.
υ The VNB is named after John von Neumann, a computer scientist who was credited with the invention of the bus based computer architecture.
υ To allow faster memory access, various distributed memory "non-von" systems were proposed

➢ **Buffer Registers:** ▫ The devices connected to a bus vary widely in their speed of operation. ▫ To synchronize their operational-speed, buffer-registers can be used ▫ are included with the devices to hold the information during transfers. ▫ prevent a high-speed processor from being locked to a slow I/O device during data transfers.

Basic operational Concepts:
These five units work together in a cycle called the fetch-decode-execute cycle:
Fetch: The control unit retrieves an instruction from memory.
Decode: The control unit breaks down the instruction into its components (operation, operands) and sends them to the ALU.
Execute: The ALU performs the operation on the operands and stores the result back in memory or sends it to the output unit.
Register Transfer
Definition: The process of moving data between registers within a computer system, under the control of the control unit.
Purpose: To enable data manipulation, storage, and retrieval for various operations.
Key Components:
Registers: Small, high-speed storage units within the processor.
Buses: Data pathways connecting registers and other components.
Control Unit: Orchestrates data flow and register operations.
Register Transfer Language (RTL):
Symbolic notation for describing register transfers.
Used for:
Modeling hardware behavior at a detailed level.
Designing digital circuits and systems.
Verifying correctness of hardware designs.
Example: `R1 ← R2 + R3` (Add contents of R2 and R3, store result in R1)
Types of Register Transfer Operations:
Register-to-register transfer: Data moves between registers.
Register-to-memory transfer: Data moves between a register and memory.
Memory-to-register transfer: Data moves from memory to a register.
Arithmetic operations: Performed on data in registers (e.g., addition, subtraction).
Logical operations: Performed on data in registers (e.g., AND, OR, NOT).
Shift operations: Shift data bits within a register (e.g., left shift, right shift).
What is Register Transfer Language (RTL)?
A symbolic notation used to describe the micro-operations that move data between registers within a digital system.
Provides a concise and precise way to model the hardware-level behavior of a system, independent of specific hardware implementation.
It's like a language that hardware designers use to communicate and document their designs.
RTL Syntax:
Uses symbols to represent registers, operations, and control signals.
Example: `R1 ← R2 + R3` (Add the contents of registers R2 and R3, and store the result in R1)
Memory Transfer Operations: Refer to the fundamental processes of reading data from memory and writing data to memory.
Common Notation:
`DR ← M[AR]` (Read operation): Transfers data from memory location specified by address register (AR) to data register (DR).
`M[AR] ← DR` (Write operation): Transfers data from data register (DR) to memory location specified by AR.
Arithmetic micro-operations:
• Some of the basic micro-operations are addition, subtraction, increment and decrement.
➢ **Add Micro-Operation:**
▫ It is defined by the following statement: R3 → R1 + R2 ▫ The above statement instructs the data or contents of register R1 to be added to data or content of register R2 and the sum should be transferred to register R3.
➢ **Subtract Micro-Operation:**
▫ Let us again take an example: R3 → R1 + R2' + 1 ▫ In subtract micro-operation, instead of using minus operator we take 1's compliment and add 1 to the register which gets subtracted, i.e R1 - R2 is equivalent to R3 → R1 + R2' + 1
➢ **Increment/Decrement Micro-Operation:**
▫ Increment and decrement operation are generally performed by adding and subtracting 1 to and from the register respectively. [R1 → R1 + 1R1 → R1 – 1] Symbolic Designation Description R3 ← R1 + R2 Contents of R1+R2 transferred to R3. R3 ← R1 - R2 Contents of R1-R2 transferred to R3. R2 ← (R2)' Compliment the contents of R2. R2 ← (R2)' + 1 2's compliment the contents of R2. R3 ← R1 + (R2)' + 1 R1 + the 2's compliment of R2 (subtraction). R1 ← R1 + 1 Increment the contents of R1 by 1. R1 ← R1 - 1 Decrement the contents of R1 by 1. ❑**Logic micro-operations:**
• These are binary micro-operations performed on the bits stored in the registers. These operations consider each bit separately and treat them as binary variables. • Let us consider the X-OR micro-operation with the contents of two registers R1 and R2. P: R1 ← R1 X-OR R2 • In the above statement we have also included a Control Function. • Assume that each register has 3 bits. Let the content of R1 be 010 and R2 be 100. The XOR micro-operation will be:
❑**Shift micro-operations:**
• These are used for serial transfer of data. That means we can shift the contents of the register to the left or right. In the shift left operation the serial input transfers a bit to the right most position and in shift right operation the serial input transfers a bit to the left most position. • There are three types of shifts as follows:
a) Logical Shift:
▫ It transfers 0 through the serial input. The symbol "shl" is used for logical shift left and "shr" is used for logical shift right. R1 ← she R1R1 ← she R1 ▫ The register symbol must be same on both sides of arrows.
b) Circular Shift:
▫ This circulates or rotates the bits of register around the two ends without any loss of data or contents. In this, the serial output of the shift register is connected to its serial input. ▫ "cil" and "cir" is used for circular shift left and right respectively
c) Arithmetic Shift:
▫ This shifts a signed binary number to left or right. ▫ An arithmetic shift left multiplies a signed binary number by 2 and shift left divides the number by 2.
▫ Arithmetic shift micro-operation leaves the sign bit unchanged because the signed number remains same when it is multiplied or divided by 2.
▫ An left arithmetic shift operation must be checked for the overflow

Micro Programmed Control:
The function of the control unit in a digital computer is to initiate sequence of microoperations.
Control unit can be implemented **in two** ways :
- Hardwired control
- Microprogrammed control

Hardwired Control:
When the control signals are generated by hardware using conventional logic design techniques, the control unit is said to be hardwired.
The key characteristics are
High speed of operation
Expensive
Relatively complex
No flexibility of adding new instructions
Examples of CPU with hardwired control unit are Intel 8085, Motorola 6802, Zilog 80, and any RISC CPUs.
Microprogrammed Control:
Control information is stored in control memory.
Control memory is programmed to initiate the required sequence of micro-operations.
The key characteristics are
Speed of operation is low when compared with hardwired
Less complex
Less expensive
Flexibility to add new instructions
Examples of CPU with microprogrammed control unit are Intel 8080, Motorola 68000 and any CISC CPUs.
What is a microprogrammed control unit (MCU)?
An MCU is a type of control unit that uses a microprogram to control the operation of the processor.
MCUs are generally slower than hardwired control units, but they offer advantages like flexibility and easier programmability.
What is a microprogram?
A microprogram is a set of low-level instructions that specify the exact sequence of micro-operations needed to execute a single machine instruction.
Each micro-operation involves activating specific control signals within the processor to perform tasks like fetching
data, decoding instructions, or performing arithmetic operations.
What is control memory?
Control memory is a type of read-only memory (ROM) that stores the microprogram.
During the fetch-decode-execute cycle, the control unit retrieves the appropriate microinstruction from control memory based on the current machine instruction being executed.
The microinstruction then provides the control signals necessary to perform the next micro-operation.
Key characteristics of control memory:
Read-only: The microprogram in control memory is typically fixed and cannot be modified dynamically.
Fast access: Control memory needs to be fast enough to keep up with the execution speed of the processor.
Limited capacity: Control memory typically stores only the microprograms for frequently used instructions. Less common instructions may be decoded and executed using routines in main memory.
Benefits of using control memory:
Flexibility: The microprogram can be easily changed to support new instructions or modify existing functionality.
Modular design: Microprograms can be broken down into smaller, manageable units.
Cost-effective: MCUs with control memory can be cheaper to manufacture than hardwired control units.
Address sequencing
Address sequencing is a fundamental concept in computer architecture that refers to the process of determining the next address in the control memory where the next microinstruction for executing a machine instruction is stored. It's like following a roadmap to navigate through the microprogram stored in control memory.
Purpose:
Defines the order in which microinstructions are fetched from control memory to execute a machine instruction.
Ensures smooth and efficient execution of the machine instruction by fetching the right microinstructions at the right time.
Capabilities:
Incrementing the Control Address Register (CAR): This is the most basic capability, where the address in the CAR is simply incremented to fetch the next microinstruction in sequence.
Conditional Branching: Based on the outcome of a previous operation (e.g., the value of a status register), the address sequencing can jump to a different location in the control memory to fetch the next microinstruction. This allows for conditional execution of different parts of the microprogram.
Unconditional Branching: This is similar to conditional branching, but the jump to a different location happens regardless of any condition. It's often used for loops or subroutine calls.
Subroutine Calls and Returns: Address sequencing facilitates calling subroutines, which are smaller routines that can be used repeatedly within a program. It keeps track of the return address

so that after the subroutine execution, the control flow can return to the main program.

Importance:

Efficient address sequencing is crucial for optimal performance of the processor. Any delays or errors in fetching the correct microinstructions can significantly slow down the execution of the machine instruction.

It enables complex operations by breaking down machine instructions into smaller, manageable microinstructions and fetching them in the correct order.

The design of a control unit

involves defining the hardware and logic necessary to sequence and orchestrate the execution of instructions within a processor. It plays a crucial role in directing the flow of data, activating various units in the processor, and ultimately determining the behavior of the computer system.

Components:

Instruction Register (IR): Stores the currently fetched instruction being executed.

Program Counter (PC): Points to the address of the next instruction to be fetched.

Decoder: Decodes the opcode (operation code) in the instruction to determine the operation to be performed.

Control Logic: Generates control signals based on the decoded opcode and other inputs like flags and interrupt signals.

Sequential Logic: Responsible for updating the PC and fetching the next instruction after completion of the current one.

Control Memory (MCU only): Stores the microprogram, a sequence of microinstructions detailing the steps for executing each machine instruction.

What is Pipelining:

In computer architecture, pipelining refers to a technique for improving instruction execution speed by overlapping the execution stages of different instructions. Imagine it like an assembly line in a factory, where multiple stages of production happen simultaneously on different products

.

Here's how pipelining works:

Instruction Fetch: The processor fetches an instruction from memory.

Instruction Decode: The instruction is decoded to determine the operation to be performed and the operands needed.

Operand Fetch: The operands (data) required for the operation are fetched from memory or registers.

Execution: The ALU (Arithmetic Logic Unit) performs the operation on the operands.

Write Back: The result of the operation is written back to a register or memory location

Benefits of Pipelining:

Increased processor speed: Pipelining significantly improves instruction throughput, potentially doubling or even tripling the execution speed compared to a non-pipelined processor.

Improved efficiency: Resources are used more effectively, reducing idle time and maximizing the utilization of the processor's components.

Challenges/limitations of Pipelining:

Increased complexity: Pipelined processors require more complex hardware and control logic to manage the overlaps and potential hazards (data dependencies between instructions).

Pipeline hazards: In certain situations, instructions waiting in the pipeline may have to stall or be flushed due to data dependencies, which can reduce the overall speedup.

what is Instruction Pipeline:

Instruction pipelining is a specific type of pipelining used in computer architecture to improve the speed of instruction execution by dividing the instruction cycle into smaller, overlapping stages that run concurrently. Think of it like an assembly line in a factory, where different parts of the same instruction are processed simultaneously on different "stations" within the processor.

what is Arithmetic Pipeline:

An arithmetic pipeline is a technique used in computer architecture to improve the performance of arithmetic operations, particularly multiplication and floating-point calculations. It works by dividing the operation into smaller, overlapping stages that can be executed concurrently on different units within the processor. This is similar to how an assembly line in a factory works, where different parts of the same product are processed simultaneously on different stations.

RISC architecture :

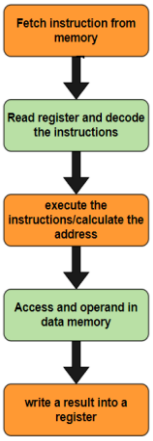
Reduced Instruction Set Computer is a special kind of Instruction Set Architecture with attributes with lower cycles per Instruction (CPI) than CISC.

RISC is a load/ store architecture as memory is only accessed through specific instructions rather than as a part of most instructions.

RISC architecture is widely used across various platforms, from cellular telephones to the fastest supercomputer

RISC Pipeline:

Pipelining, a (standard feature in RISC processors) is like an assembly line. Because the processor function on different steps of the Instruction at the same time, more instructions can be operated/executed in a short time. Several steps vary in different processors, but that steps are generally variations of these five:



Pipelining is used to improve overall performance.

Features of the RISC pipeline

RISC pipeline can use many registers to decrease the processor memory traffic and enhance operand referencing.

It keeps the most frequently accessed operands in the CPU registers.

In the RISC pipeline, simplified instructions are used, leaving complex instructions.

Here register to memory operations is reduced.

Instructions take a single clock cycle to get executed.

Example

Let's consider Instruction in the circumstances related to RISC architecture. In RISC machines, registering is most of the operations. Therefore, the instructions can be performed in two phases:

E: Execute Instruction on register operands and keep/store the results in the register.

F: Instruction Fetch to get the Instruction.

Generally, the memory access in RISC is performed through STORE and LOAD operations. For these types of instructions, the following steps are required:

F: Fetch instructions to get the Instruction

E: Effective address calculation for the required memory operand

D: register-to-memory or memory-to-register data transfer through the bus

Importance of RISC

The importance of RISC processors is as follows:

- Register-based execution
- Fixed Length instruction and Fixed Instruction Format
- Few Powerful Instructions
- Hardwired control unit
- Highly Pipelined Superscalar Architecture
- Highly Integrated Architecture

Advantages of RISC

- This RISC architecture allows developers the freedom to make use of the space on the microprocessor.
- RISC allows high-level language compilers to generate efficient code due to the architecture having a set of instructions.
- RISC processors utilize only a few parameters; besides, RICS processors cannot call instructions; hence, it uses fixed-length instructions that are easy to pipeline.
- RISC reduces the execution time while increasing the overall operation speed and efficiency.
- RISC is relatively simple because it has very few instruction formats; also, a small number of instructions and a small number of addressing modes are needed.

what is Vector Processing:

Vector processing is a technique used in computer architecture to improve the performance of computations that involve operating on large arrays of data simultaneously. Instead of processing each element of the array individually, vector processors can apply the same operation to all elements in parallel, significantly boosting speed. Think of it like processing a bunch of apples on a conveyer belt instead of doing them one by one.

vector processing works:

Fetch Vector Instructions: The processor fetches an instruction that specifies the operation to be performed on the vector (array) of data.

Load Vector Data: The vector data is loaded from memory into special registers within the processor called vector registers. These registers can hold multiple data elements, unlike regular registers which hold only one.

Execute Vector Operation: The ALU (Arithmetic Logic Unit) performs the specified operation on all elements of the vector data simultaneously using specialized parallel processing units.

Store Vector Result: The result of the operation is stored back to a vector register or memory location.

Benefits of Vector Processing:

Increased Performance: Vector processing can significantly improve the speed of computations that involve large arrays of data, especially for operations like addition, subtraction, multiplication, and other basic arithmetic operations.

Improved Efficiency: By processing multiple data elements simultaneously, vector processors can utilize the processor's resources more effectively, reducing idle time and maximizing throughput.

Reduced Programming Complexity: Vector instructions can simplify the code for performing repetitive operations on large arrays, making it easier for programmers to express these types of computations.

Challenges of Vector Processing:

Increased Hardware Complexity: Vector processors require specialized hardware, such as vector registers and parallel processing units, which can increase the cost and complexity of the processor.

Not all algorithms benefit: Not all algorithms are well-suited for vector processing. Some algorithms may have dependencies between data elements that prevent them from being processed in parallel.

Memory Access Bottleneck: Accessing data in memory can be a bottleneck for vector processing, as fetching large vectors can take longer than the actual operation itself.

Applications of Vector Processing:

Scientific Computing: Vector processing is widely used in scientific computing applications that involve large datasets, such as weather forecasting, climate modeling, and computational fluid dynamics.

Image and Signal Processing: Vector processing is also used in image and signal processing applications, such as filtering, compression, and transformation of images and audio signals.

Machine Learning: Vector processing is playing an increasingly important role in machine learning applications that involve training algorithms on large datasets, such as deep learning and image recognition.

what is array Processing:

The term "array processing" can be interpreted in two different ways, depending on the context:

1. Processing data within an array:

In this context, array processing refers to the act of applying operations to all elements of an array of data simultaneously. This can involve simple operations like addition or multiplication, as well as more complex operations like filtering, sorting, or performing statistical analyses.

Array processing can be done in different ways:

Using traditional loops: This is the most basic approach, where you iterate through each element of the array individually and apply the desired operation. However, this can be slow and inefficient for large arrays.

Using specialized libraries or frameworks: Many programming languages and libraries offer optimized functions for performing common operations on arrays. These functions use vectorization and other techniques to improve performance.

Using parallel processing techniques: If you have multiple processors or cores available, you can distribute the work of processing the array across them. This can significantly improve performance for large arrays.

2. Array processors:

In a different context, "array processing" can also refer to a specific type of hardware processor designed to efficiently handle computations involving large arrays of data. These processors typically have special features like:

Vector registers: These registers can hold multiple data elements from an array, allowing for faster access and manipulation compared to regular registers.

Parallel processing units: These units can perform the same operation on all elements of a vector simultaneously, further improving performance.

Specialized instructions: Array processors often have instructions designed specifically for manipulating arrays, such as vector addition, multiplication, and sorting.

Examples of applications that benefit from array processing include:

Scientific computing: Calculations involving large datasets in areas like weather forecasting, climate modeling, and fluid dynamics.

Image and signal processing: Operations like filtering, compression, and transformation of images and audio signals.

Machine learning: Training algorithms on large datasets for tasks like natural language processing and image recognition.

An instruction set, sometimes called an Instruction Set Architecture (ISA), is the fundamental collection of instructions that a microprocessor can understand and execute. It's like a dictionary defining the commands the processor can interpret and the operations it can perform. Understanding the instruction set is crucial for programmers and computer architects as it determines the capabilities and limitations of the processor.

Here are some key aspects of microprocessor instruction sets:

Components of an Instruction:

Opcode: This is the code that identifies the specific operation to be performed.

Operands: These are the data elements involved in the operation, such as register numbers or memory addresses.

Addressing modes: These specify how the operands are located in memory or registers.

Flags: These are status bits that indicate the outcome of previous operations, like carry or overflow.

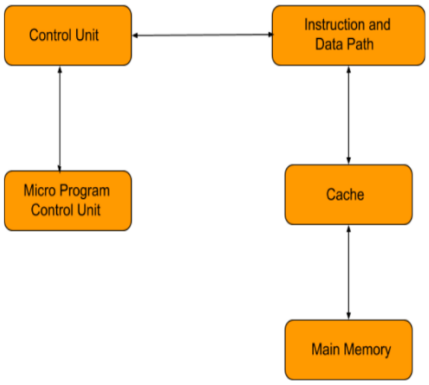
Categories of Instructions:

Arithmetic and logic instructions: These perform basic operations like addition, subtraction, multiplication, and comparisons.

Data transfer instructions: These move data between registers, memory, and input/output devices.

Control flow instructions: These control the flow of execution within a program, including loops, branches, and subroutine calls.
Processor control instructions: These manage the internal state of the processor, such as setting flags or enabling/disabling interrupts.

Types of Instruction Sets:
Complex Instruction Set Architecture (CISC): Offers a large and diverse set of instructions, often tailored for specific tasks. Examples include x86 and ARM.
Examples of CISC processors are AMD, Intel x86, and the System/360.
CISC Architecture:

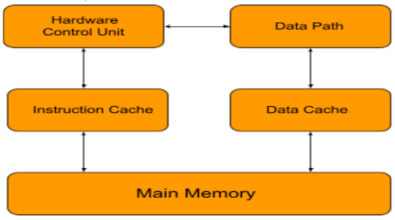


Features of CISC Processor:
- CISC may take longer than a single clock cycle to execute the code.
- The length of the code is short, so it requires minimal RAM.
- It provides more accessible programming in assembly language.
- It focuses on creating instructions on hardware rather than software because they are faster to develop.
- It comprises fewer registers and more addressing nodes, typically 5 to 20.

Difference Between RISC And CISC:

RISC	CISC
It is a Reduced Instruction Set Computer.	It is a Complex Instruction Set Computer.
It focuses on Software.	It focuses on Hardware.
It uses only a Hardwired control unit.	It uses both hardwired and microprogrammed control units.
Transistors are used for more registers.	Transistors are used for storing complex instructions.
Code size is large.	Code size is small.
The uses of the pipeline are simple in RISC.	Uses of the pipeline are difficult in CISC.
An instruction is executed in a single clock cycle.	Instruction may take more than one clock cycle.
An instruction can fit in one word.	Instructions are larger than the size of one word.
The execution time of RISC is very short.	The execution time of CISC is longer.
The program written for RISC architecture needs to take more space in memory.	The Program written for CISC architecture tends to take less space in memory.

Reduced Instruction Set Architecture (RISC): Employs a smaller set of simpler, core instructions that are frequently used. Examples include MIPS and PowerPC.



Examples of RISC processors are PowerPC, Microchip PIC, SUN's SPARC, RISC-V.
Features of RISC Processor :
- RISC processors use one clock per cycle (CPI) to execute each instruction in a computer. Each CPI also comprises the methods for fetching, decoding, and executing computer instructions.
- Multiple registers in RISC processors allow them to hold instructions, reply fast to the computer, and interact with computer memory as little as possible.
- The RISC processors use the pipelining technique to execute multiple parts or stages of instructions to perform more efficiently.
- RISC has a simple addressing mode and fixed instruction length for the pipeline execution.
- It uses LOAD and STORE instruction to access the memory location.

Relatively used few instructions and few addressing modes. It is Hardwired rather than micro programmed control

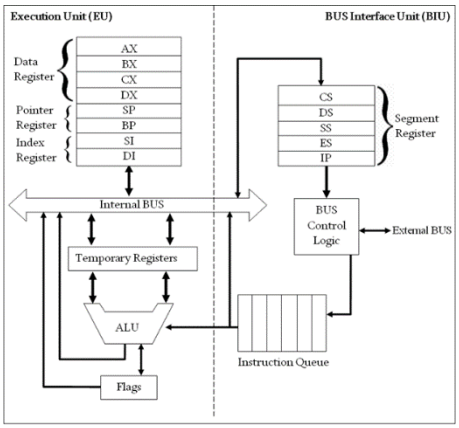
Instruction Set Design Issues

> Where are operands stored? - registers, memory, stack, accumulator
> How many explicit operands are there? - 0, 1, 2, or 3
How is the operand location specified? - register, immediate, indirect, . . .
> What type & size of operands are supported? - byte, int, float, double, string, vector. . .
> What operations are supported? - add, sub, mul, move, compare . .

Registers Advantages and Disadvantages
Advantages:
Faster than cache or main memory (no addressing mode)
Deterministic (no misses)
Can replicate (multiple read ports)
Short identifier (typically 3 to 8 bits)
Reduce memory traffic
Disadvantages:
Need to save and restore on procedure calls and context switch
Can't take the address of a register (for pointers)
Fixed size (can't store strings or structures efficiently) Compiler must manage

Introduction of Intel 8086:
• Intel 8086 microprocessor is the enhanced version of Intel 8085 microprocessor. It was designed by Intel in 1976.
• The 8086 microprocessor is a16-bit, N-channel, HMOS microprocessor. Where the HMOS is used for "High-speed Metal Oxide Semiconductor".
• Intel 8086 is built on a single semiconductor chip and packaged in a 40-pin IC package. The type of package is DIP (Dual Inline Package).
•intel 8086 uses 20 address lines and 16 data- lines. It can directly address up to 220 = 1 Mbyte of memory.
• It consists of a powerful instruction set, which provides operation like division and multiplication very quickly.
• 8086 is designed to operate in two modes, i.e., Minimum and Maximum mode.

Block Diagram of 8086



8086 contains two independent functional units: **a Bus Interface Unit (BIU)** and an **Execution Unit (EU)**.
Bus Interface Unit (BIU):
The segment registers, instruction pointer and 6-byte instruction queue are associated with the bus interface unit (BIU).
It handles transfer of data and addresses,
It Fetches instruction codes, stores fetched instruction codes in first-in-first-out register set called a **queue**,
It Reads data from memory and I/O devices,
It Writes data to memory and I/O devices,
It relocates addresses of operands since it gets un-relocated operand addresses from EU. The EU tells the BIU from where to fetch instructions or where to read data.
It has the following functional parts:
Instruction Queue: When EU executes instructions, the BIU gets 6-bytes of the next instruction and stores them in the instruction queue and this process is known as instruction pre fetch. This process increases the speed of the processor.
Segment Registers: A segment register contains the addresses of instructions and data in memory which are used by the processor to access memory locations. It points to the starting address of a memory segment currently being used.
There are 4 segment registers in 8086 as given below:
Code Segment Register (CS): Code segment of the memory holds instruction codes of a program.
Data Segment Register (DS): The data, variables and constants given in the program are held in the data segment of the memory.
Stack Segment Register (SS): Stack segment holds addresses and data of subroutines. It also holds the contents of registers and memory locations given in PUSH instruction.
Extra Segment Register (ES): Extra segment holds the destination addresses of some data of certain string instructions.
Instruction Pointer (IP): The instruction pointer in the 8086 microprocessor acts as a program counter. It indicates to the address of the next instruction to be executed.
Execution Unit (EU):
The **EU** receives opcode of an instruction from the queue, decodes it and then executes it. While Execution, unit decodes or executes an instruction, then the BIU fetches instruction codes from the memory and stores them in the queue.
The BIU and EU operate in parallel independently. This makes processing faster.
General purpose registers, stack pointer, base pointer and index registers, ALU, flag registers (FLAGS), instruction decoder and timing and control unit constitute execution unit (EU). Let's discuss them:
General Purpose Registers: There are four 16-bit general purpose registers: AX (Accumulator Register), BX (Base Register), CX (Counter) and DX. Each of these 16-bit registers are further subdivided into 8-bit registers as shown below:

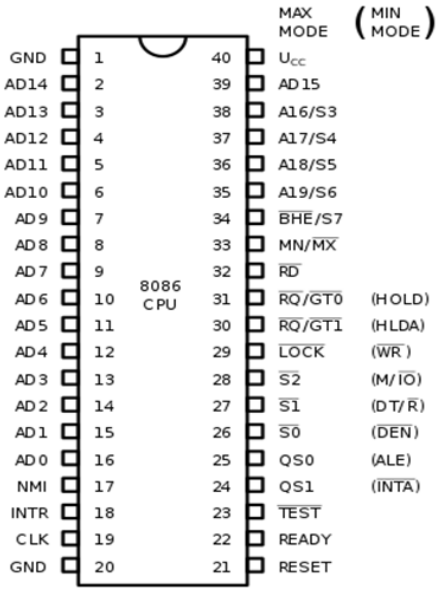
16-bit registers	8-bit high-order registers	8-bit low-order registers
AX	AH	AL
BX	BH	BL
CX	CH	CL
DX	DH	DL

Index Register: The following four registers are in the group of pointer and index registers:
- Stack Pointer (SP)
- Base Pointer (BP)
- Source Index (SI)
- Destination Index (DI)

ALU: It handles all arithmetic and logical operations. Such as addition, subtraction, multiplication, division, AND, OR, NOT operations.
Flag Register: It is a 16?bit register which exactly behaves like a flip-flop, means it changes states according to the result stored in the accumulator. It has 9 flags and they are divided into 2 groups i.e. conditional and control flags.
Conditional Flags: This flag represents the result of the last arithmetic or logical instruction executed. Conditional flags are:
- Carry Flag
- Auxiliary Flag
- Parity Flag
- Zero Flag
- Sign Flag
- Overflow Flag

Control Flags: It controls the operations of the execution unit. Control flags are:
- Trap Flag
- Interrupt Flag
- Direction Flag

Pins Diagram and Description of 8086:



AD0-AD15 (Address Data Bus): Bidirectional address/data lines. These are low order address bus. They are multiplexed with data. When these lines are used to transmit memory address, the symbol A is used instead of AD, for example, A0- A15.
A16 - A19 (Output): High order address lines. These are multiplexed with status signals.
A16/S3, A17/S4: A16 and A17 are multiplexed with segment identifier signals S3 and S4.
A18/S5: A18 is multiplexed with interrupt status S5.

A19/S6: A19 is multiplexed with status signal S6.
BHE/S7 (Output): Bus High Enable/Status. During T1, it is low. It enables the data onto the most significant half of data bus, D8-D15. 8-bit device connected to upper half of the data bus use BHE signal. It is multiplexed with status signal S7. S7 signal is available during T3 and T4.
RD (Read): For read operation. It is an output signal. It is active when LOW.
Ready (Input): The addressed memory or I/O sends acknowledgment through this pin. When HIGH, it denotes that the peripheral is ready to transfer data.
RESET (Input): System reset. The signal is active HIGH.
CLK (input): Clock 5, 8 or 10 MHz.
INTR: Interrupt Request.
NMI (Input): Non-maskable interrupt request.
TEST (Input): Wait for test control. When LOW the microprocessor continues execution otherwise waits.
VCC: Power supply +5V dc.
GND: Ground.

Operating Modes of 8086:
There are two operating modes of operation for Intel 8086, namely the **minimum mode** and the **maximum mode**.
When only one 8086 CPU is to be used in a microprocessor system, the 8086 is used in the **Minimum mode** of operation.
In a multiprocessor system 8086 operates in the **Maximum mode**.

Pin Description for Minimum Mode:
In this minimum mode of operation, the pin MN/MX is connected to 5V D.C. supply i.e. MN/MX = VCC.
The description about the pins from 24 to 31 for the minimum mode is as follows:
INTA (Output): Pin number 24 interrupts acknowledgement. On receiving interrupt signal, the processor issues an interrupt acknowledgment signal. It is active LOW.
ALE (Output): Pin no. 25. Address latch enable. It goes HIGH during T1. The microprocessor 8086 sends this signal to latch the address into the Intel 8282/8283 latch.
DEN (Output): Pin no. 26. Data Enable. When Intel 8287/8286 octal bus transceiver is used this signal. It is active LOW.
DT/R (output): Pin No. 27 data Transmit/Receives. When Intel 8287/8286 octal bus transceiver is used this signal controls the direction of data flow through the transceiver. When it is HIGH, data is sent out. When it is LOW, data is received.
M/IO (Output): Pin no. 28, Memory or I/O access. When this signal is HIGH, the CPU wants to access memory. When this signal is LOW, the CPU wants to access I/O device.
WR (Output): Pin no. 29, Write. When this signal is LOW, the CPU performs memory or I/O write operation.
HLDA (Output): Pin no. 30, Hold Acknowledgment. It is sent by the processor when it receives HOLD signal. It is active HIGH signal. When HOLD is removed HLDA goes LOW.
HOLD (Input): Pin no. 31, Hold. When another device in microcomputer system wants to use the address and data bus, it sends HOLD request to CPU through this pin. It is an active HIGH signal.

Pin Description for Maximum Mode:
In the maximum mode of operation, the pin MN/¯MX is made LOW. It is grounded. The description about the pins from 24 to 31 is as follows:
QS1, QS0 (Output): Pin numbers 24, 25, Instruction Queue Status.
S0, S1, S2 (Output): Pin numbers 26, 27, 28 Status Signals. These signals are connected to the bus controller of Intel 8288. This bus controller generates memory and I/O access control signals.
LOCK (Output): Pin no. 29. It is an active LOW signal. When this signal is LOW, all interrupts are masked and no HOLD request is granted. In a multiprocessor system all other processors are informed through this signal that they should not ask the CPU for relinquishing the bus control.
RG/GT1, RQ/GT0 (Bidirectional): Pin numbers 30, 31, Local Bus Priority Control. Other processors ask the CPU by these lines to release the local bus.
In the maximum mode of operation signals WR, ALE, DEN, DT/R etc. are not available directly from the processor. These signals are available from the controller 8288.

Register Structure of 8086:
The 8086 microprocessor uses a segmented memory architecture and has several specific registers with designated functions. Here's a breakdown of the Register Structure of 8086:
General-Purpose Registers (8 x 16 bits):
AX (Accumulator): Main arithmetic and data register, often used for temporary storage.
AL (Lower byte): Directly accessed for byte operations.
AH (Upper byte): Used for higher-order operations and specific instructions.
BX (Base Register): Used for addressing data in memory using the base pointer.
CX (Counter Register): Used for loop counters and string operations.
DX (Data Register): Used for data manipulation and I/O operations.
SP (Stack Pointer): Points to the top of the stack, used for storing return addresses and temporary data.
BP (Base Pointer): Used for addressing data in memory relative to the stack segment.
SI (Source Index): Used for indexed addressing of data in memory.
DI (Destination Index): Used for indexed addressing of data in memory during string operations.
Segment Registers (4 x 16 bits):
CS (Code Segment Register): Points to the beginning of the current code segment.
DS (Data Segment Register): Points to the beginning of the current data segment.
SS (Stack Segment Register): Points to the beginning of the current stack segment.
ES (Extra Segment Register): Optional segment register, mainly used for additional data segments.
Flag Register (1 x 16 bits):
Contains various flags indicating the state of the processor after an instruction is executed, such as carry, zero, parity, etc

Interrupt Mechanism in 8086 Microprocessor
The 8086 microprocessor utilizes interrupts to handle asynchronous events while executing the main program. This allows peripheral devices, external signals, or internal conditions to temporarily halt the current program and prioritize the urgent event before resuming the original execution.
Key Components:
Interrupt Requests (IRQs): Signals sent by devices or the CPU itself signifying the need for immediate attention. The 8086 has two main IRQ pins: INTR and NMI.
Interrupt Service Routine (ISR): A dedicated sub-program designed to handle the specific event triggered by the interrupt.
Interrupt Descriptor Table (IDT): A data structure holding information about available interrupts, including the memory address of the corresponding ISR for each interrupt number.

Benefits of Interrupts:
Enhance responsiveness to external events and improve multitasking capabilities.
Prioritize urgent tasks without halting the entire program execution.
Efficiently handle asynchronous events without polling, reducing processor overhead.

Addressing modes in the 8086 microprocessor
define how the operand for an instruction is located in memory. By understanding these modes, you can effectively write assembly code for the 8086.
key addressing modes:
1. Immediate Addressing:
The operand is directly encoded within the instruction itself.
Example: `MOV AX, 5` - This moves the immediate value 5 into the AX register.
2. Register Addressing:
The operand is directly represented by a register.
Example: `ADD BX, DX` - This adds the contents of the DX register to the BX register.
3. Direct Addressing:
The operand's address is explicitly encoded within the instruction using a 16-bit offset.
Example: `MOV EAX, [1000]` - This moves the data at memory address 1000 into the EAX register.
4. Register Indirect Addressing:
The operand's address is stored in a specific register.
Example: `MOV EAX, [BX]` - This moves the data at the memory address stored in the BX register into the EAX register.
5. Indexed Addressing:
The operand's address is calculated by adding a register value (SI or DI) to a base address.
Example: `MOV ECX, [BX + SI]` - This moves the data at the memory address stored in BX + SI into the ECX register.
6. Based Addressing:
The operand's address is calculated by adding a base register value (BP) to an offset.
Example: `MOV DX, [300 + BP]` - This moves the data at the memory address 300 + BP into the DX register.
7. Based Indexed Addressing:
Combines both based and indexed addressing, adding a base register (BP), an index register (SI or DI), and an offset.
Example: `MOV AX, [BX + SI + 50]` - This moves the data at the memory address BX + SI + 50 into the AX register.
8. Inter-segment Addressing:
Accesses data in a different segment than the current one using segment registers.
Requires additional instructions and prefix bytes for effective access.

Instruction Set Categories of 8086:
1. Data Transfer Instructions:
Move data between registers, memory, and I/O ports.
Examples: MOV, PUSH, POP, XCHG, IN, OUT
2. Arithmetic Instructions:
Perform mathematical operations like addition, subtraction, multiplication, division, increment, decrement, and comparison.
Examples: ADD, SUB, MUL, DIV, INC, DEC, CMP
3. Bit Manipulation Instructions:
Work with individual bits within bytes or words.
Examples: AND, OR, XOR, NOT, SHL, SHR, ROL, ROR, TEST
4. String Instructions:
Handle operations on strings (sequences of bytes or words).
Examples: MOVS, CMPS, SCAS, LODS, STOS
5. Program Execution Transfer Instructions:
Control the flow of program execution.
Examples: JMP, CALL, RET, LOOP, JCXZ
6. Processor Control Instructions:
Manage the processor's state and operations.
Examples: HLT, NOP, STC, CLC, CMC, STD, CLD, STI, CLI
7. Flag Manipulation Instructions:
Set or clear the status flags in the flag register.
Examples: STC, CLC, CMC, STD, CLD, STI, CLI

1. What is an opcode?
The part of the instruction that specifies the operation to be performed is called the operation code or opcode.
2. What is an operand?
The data on which the operation is to be performed is called as an operand.
3. What is meant by wait state?
This state is used by slow peripheral devices. The peripheral devices can transfer the data to or from the microprocessor by using READY input line. The microprocessor remains in the wait state as long as READY line is low. During the wait state, the contents of the address, address/data and control buses are held constant.
4. What are the functions of an accumulator?
The accumulator is the register associated with the ALU operations and sometimes I/O operations. It is an integral part of ALU. It holds one of data to be processed by ALU. It also temporarily stores the result of the operation performed by the ALU.
5. What is meant by polling?
Polling or device polling is a process which identifies the device that has interrupted the microprocessor.
6. What is meant by interrupt?
Interrupt is an external signal that causes a microprocessor to jump to a specific subroutine.
7. Define instruction cycle, machine cycle and T-state?
Instruction cycle is defined as the time required completing the execution of an instruction. Machine cycle is defined as the time required completing one operation of accessing memory, I/O or acknowledging an external request. T cycle is defined as one subdivision of the operation performed in one clock period.
8. Explain the signals HOLD, READY and SID.
HOLD indicates that a peripheral such a DMA controller is requesting the use of address bus, data bus and control bus.
READY is used to delay the microprocessor read or write cycles until a slow responding peripheral is ready to accept or send data.
SID is used to accept serial data bit by bit.
9. What is interfacing?
An interface is a shared boundary between the devices which involves sharing information. Interfacing is the process of making two different systems communicate with each other.
10. **What is memory mapping?**
The assignment of memory address to various registers in a memory chip is called as memory mapping.

Assembly Language
Assembly language, often abbreviated as ASM, is a low-level programming language that bridges the gap between the hardware of a computer and the high-level languages programmers typically use. Unlike its more beginner-friendly counterparts like Python or Java, assembly language instructions directly correspond to the machine code understood by the CPU. This means that assembly code provides fine-grained control over the hardware, but at the cost of being much more difficult to write and understand for humans.
example of a simple assembly language instruction:
`MOV AX, BX`

Why Use Assembly Language?
Performance: Assembly code can be highly optimized for specific hardware, leading to significantly faster execution compared to high-level languages. This makes it attractive for performance-critical applications like operating systems, device drivers, and embedded systems.
Direct Hardware Access: Assembly language allows programmers to directly interact with the hardware components of a computer, giving them fine-grained control over things like memory management and peripheral devices.
Understanding Computer Architecture: Learning assembly language can provide a deeper understanding of how computers work at the fundamental level, which can be valuable for software engineers and hardware developers.
However, assembly language also has its drawbacks:
Complexity: As mentioned earlier, assembly language is much more difficult to write and understand than high-level languages. This makes it less accessible to beginners and requires specialized knowledge of the specific CPU architecture being used.
Error-prone: The low-level nature of assembly language makes it more prone to errors, as even small mistakes can have significant consequences. Debugging assembly code can be a challenging task.
Platform-specific: Assembly language instructions are specific to the underlying CPU architecture. This means that code written for one processor won't necessarily work on another, limiting its portability.

Assembler: The Bridge Between Humans and Computers
Imagine a translator who can turn your everyday words into the intricate, low-level language a computer understands. That's essentially what an assembler does! It's a special program that takes instructions written in assembly language, a language closer to the inner workings of the computer, and translates them into machine code, the binary language directly understood by the processor.
some key assembler directives used in 8086 assembly language:
Segment Directives:
SEGMENT and ENDS: Define the beginning and end of a memory segment (code, data, stack, extra).
ASSUME: Informs the assembler about the intended segment register for a given memory reference.
Procedure Directives:
PROC and ENDP: Define the start and end of a procedure (a reusable block of code).
Macro Directives:
MACRO and ENDM: Define a macro, which is a template for generating multiple instructions with different parameters.
Data Definition Directives:
DB (Define Byte): Allocates one or more bytes of memory and optionally initializes them with values.
DW (Define Word): Allocates one or more words (2 bytes each) of memory.

DD (Define Doubleword): Allocates one or more doublewords (4 bytes each) of memory.

Pointer Directives:

PTR: Specifies the size of a memory operand (byte ptr, word ptr, dword ptr).

Assembly Control Directives:

ORG: Sets the origin of a segment, specifying a starting address in memory.

END: Marks the end of the assembly code.

Model Directives:

.**MODEL:** Specifies the memory model used (small, medium, large, compact) to determine how segments are organized.

macro

In assembly language, a MACRO is a powerful tool that allows you to create reusable code templates, effectively reducing code repetition and improving readability.

Here's how macros work:

Definition:

You define a macro using the `MACRO` directive, followed by its name and optional parameters.

The body of the macro contains the code template you want to reuse.

The `ENDM` directive marks the end of the macro definition.

Expansion:

When you invoke the macro in your code (by using its name and providing any necessary arguments), the assembler expands it inline.

It substitutes the provided arguments for the corresponding parameters in the macro body.

The expanded code is then assembled as if you had written it out manually.

Example:

This macro defines a template for printing two strings to the console. To use it:

```
PrintString "Hello,", "world!"
```

Advantages

It allows complex jobs to run in a simpler way.

It is memory efficient, as it requires less memory.

It is faster in speed, as its execution time is less.

It is mainly hardware-oriented.

It requires less instruction to get the result.

It is used for critical jobs.

It is not required to keep track of memory locations.

It is a low-level embedded system.

Disadvantages

It takes a lot of time and effort to write the code for the same.

It is very complex and difficult to understand.

The syntax is difficult to remember.

It has a lack of portability of program between different computer architectures.

It needs more size or memory of the computer to run the long programs written in Assembly Language.

Memory Devices:

1. **RAM (Random Access Memory):**

Stores data that can be read and written to at any time.

Volatile, meaning data is lost when power is turned off.

Used for temporary storage of data used by the processor during program execution.

2. **ROM (Read-Only Memory):**

Stores permanent data that can only be read, not written to.

Non-volatile, meaning data is retained even when power is turned off.

Used for storing programs and essential system data.

1. **EPROM (Erasable Programmable Read-Only Memory):**

Similar to ROM in functionality, but data can be erased using ultraviolet light and reprogrammed.

Offers flexibility for development and prototyping.

Data Access:

RAM: Read and write access, allowing for frequent data changes. Like a two-way street, data can flow both in and out.

ROM: Read-only access, typically containing pre-programmed instructions or data. Think of a one-way street, information only flows out.

EPROM: Read-only access after programming, but data can be erased and reprogrammed using a special device. Imagine a one-way street with a dedicated eraser that allows you to rewrite the information flowing out.

3. Typical Usage:

RAM: Holds temporary data used by the processor during program execution, like open applications and files. It's the workhorse for active tasks.

ROM: Stores essential system software like the BIOS and basic input/output routines (BIOS). It's the foundation for booting up and basic functionality.

EPROM: Used for development and prototyping when frequent updates to firmware or programs are needed. It's the flexible option for testing and adjustments.

4. Cost and Speed:

RAM: Generally cheaper but has slower read/write speeds compared to ROM and EPROM.

ROM: More expensive but boasts faster speeds than RAM.

EPROM: Moderate cost with read/write speeds between RAM and ROM. However, the erasing and reprogramming process can be time-consuming.

Storage Mechanism:

DRAM (Dynamic Random Access Memory):

Uses capacitors to store data. However, these capacitors leak charge over time, requiring periodic refreshing to maintain the data. It's like a leaky bucket that needs constant refilling to keep the water level stable.

SRAM (Static Random Access Memory): Utilizes flip-flops (circuits built with transistors) to store data. These circuits latch the data and don't need refreshing, making them faster and more efficient. Imagine a bucket with a tight lid that retains the water without needing refills.

Key Differences:

Speed:

SRAM: Significantly faster than DRAM due to its static storage mechanism. Access times can be 10 times faster or even more.

DRAM: Slower due to the need for refreshing, with access times typically in the range of nanoseconds compared to picoseconds for SRAM.

Power Consumption:

SRAM: Consumes more power than DRAM because the flip-flops constantly draw current to maintain the data.

DRAM: More power-efficient because of the simpler capacitor-based storage and the need for refreshing only occasionally.

Cost:

SRAM: More expensive than DRAM due to the complexity of the flip-flop circuits.

DRAM: Less expensive because of the simpler capacitor storage and denser chip design.

Applications:

SRAM: Used in situations where speed is critical, such as cache memory in processors, embedded controllers, and high-performance networking devices.

DRAM: Used for large-capacity main memory in computers and various devices due to its lower cost and adequate speed for most applications.

Cache memory

Cache memory is a small, fast memory that sits closer to the processor than the main memory (RAM). It acts as a temporary storage area for frequently accessed data and instructions, bridging the speed gap between the processor and main memory.

Think of it as a personal assistant to the processor:

The processor requests data or instructions.

The cache memory, being closer and faster, checks if it has the needed information already stored.

If it does, it quickly provides it to the processor, saving time and improving performance.

If not, it fetches the data from the slower main memory, stores it in cache for future use, and then delivers it to the processor.

Key characteristics of cache memory:

Smaller in size: Typically ranges from a few kilobytes to several megabytes, compared to gigabytes of main memory.

Much faster: Access times are often 10-100 times faster than main memory.

Costlier: Due to its speed and design, it's more expensive per byte than main memory.

Multi-level: Modern processors often have multiple levels of cache (L1, L2, L3) for even better performance.

Benefits of cache memory:

Reduces access time to data and instructions: Speeds up program execution and overall system responsiveness.

Improves overall system performance: Makes a significant difference in tasks that require frequent data access, like gaming, video editing, and web browsing.

Reduces power consumption: By minimizing the need to access main memory, it conserves energy.

Locality of reference

It is a fundamental concept in computer science that describes the tendency of a program to repeatedly access a relatively small set of memory locations within a specific timeframe. This phenomenon plays a crucial role in optimizing memory access and improving system performance.

There are two main types of locality of reference:

1. **Temporal locality:** This refers to the tendency of a program to reuse specific data or instructions repeatedly over a short period. Think of it like revisiting frequently used pages in a book. For example, a loop in a program will access the same instructions and data elements many times until the loop finishes.
2. **Spatial locality:** This describes the tendency of a program to access memory locations near recently accessed ones. Imagine browsing through chapters in a book consecutively instead of jumping around randomly. For example, accessing an array element often leads to accessing nearby elements within the same array shortly afterward.

Cache mapping

it is the strategy used to determine where data from main memory is placed within the cache memory. It's like organizing a small, efficient workspace to maximize productivity.

Here are the three primary cache mapping techniques:

1. **Direct Mapping:**

Each main memory block can only be placed in one specific block of the cache.

Simple and fast, but can lead to conflicts when multiple blocks map to the same cache block.

Imagine a bookshelf with fixed slots for certain genres: each book can only go in its designated spot.

2. Associative Mapping:

Any main memory block can be placed in any block of the cache.

Offers more flexibility and reduces conflicts, but requires more complex hardware to search for data.

Think of a desk with multiple drawers, where you can put any item in any drawer, allowing for more efficient organization.

3. Set-Associative Mapping:

A compromise between direct and associative mapping.

Cache is divided into sets, and each main memory block can be placed in any block within a specific set.

Balances flexibility with implementation complexity, reducing conflicts while maintaining a simpler hardware design.

Picture a multi-shelf unit with multiple compartments on each shelf: items can be placed in any compartment within a designated shelf, providing both structure and adaptability.

In the context of cache memory, hit ratio and miss ratio are crucial metrics for evaluating its effectiveness. They tell you how often the information needed by the processor is readily available in the cache and how often it needs to be fetched from the slower main memory.

Hit Ratio:

Represents the percentage of memory accesses that the cache successfully fulfills.

Indicates how often the information requested by the processor is already stored in the cache, saving time and improving performance.

A higher hit ratio signifies a more efficient cache, meaning the processor often finds what it needs close at hand.

Miss Ratio:

Represents the percentage of memory accesses that result in a miss, meaning the requested information is not found in the cache.

Requires the processor to fetch the data from the slower main memory, leading to a performance penalty.

A lower miss ratio is desirable, as it minimizes the need for slower memory access and improves overall system performance.

Calculation:

Hit Ratio = (Number of cache hits) / (Total number of memory accesses)

Miss Ratio = (Number of cache misses) / (Total number of memory accesses)

Physical Address:

Imagine this: your house has a unique street address, like 123 Main Street. This is the physical address in the real world.

Similarly, in a computer's memory, each byte of data has a unique physical address, a specific location in the memory chip.

This address is directly understandable by the hardware and tells it where to find the data.

Physical addresses are typically long strings of binary digits (0s and 1s) and can be difficult to remember or work with for humans.

Logical Address:

Now, think about how you typically refer to your house. Instead of using the complex street address, you might use a more relatable name, like "My home" or "123 House." This is similar to a logical address in a computer.

It's a symbolic or relative address that is easier for humans to understand and use in programs.

The program accesses data using logical addresses, and then the operating system or a special hardware unit called a Memory Management Unit (MMU) translates these logical addresses into the corresponding physical addresses.

Virtual memory

it is an ingenious memory management technique that allows a computer to use more memory than it physically has. It's like an illusionist's trick that makes a small stage appear much larger, enabling programs to run smoothly even when they require extensive memory resources.

Here's how it works:

1. **Illusion of Vast Memory**:

The operating system creates an illusion of a vast, contiguous virtual memory space for each program.

This virtual space is much larger than the actual physical memory (RAM) available.

It's like having a personal library with seemingly endless shelves, even if you only have a small room for books.

2. **Mapping and Translation:**

The operating system keeps track of which parts of the virtual memory are currently in physical RAM and which parts reside on a slower storage device, such as a hard disk or SSD.

It uses a special hardware unit called the Memory Management Unit (MMU) to translate virtual addresses used by programs into physical addresses for actual memory locations.

This process is like a librarian managing a vast collection, seamlessly bringing books from storage to reading rooms as needed.

3. **Seamless Swapping:**

When a program needs data that's not currently in RAM, the operating system automatically swaps out less-used data to the storage device and swaps in the required data from storage to RAM.

This happens behind the scenes, without the program's knowledge, maintaining the illusion of unlimited memory.

Imagine the librarian effortlessly exchanging books on shelves to make room for new requests, without readers ever noticing the swaps.

Benefits of Virtual Memory:

Increased memory capacity: Allows programs to run even if they exceed the physical RAM size.

Improved multitasking: Enables multiple programs to run concurrently without exhausting memory.

Simplified memory management: Programmers can focus on logical memory addresses, leaving physical memory management to the operating system.

Enhanced security: Can isolate programs from each other, preventing unauthorized memory access.

Cache Memory:

Focus: Speed up data access for the processor.

Size: Very small (kilobytes to megabytes).

Location: Closer to the processor than main memory (RAM).

Content: Stores frequently accessed data and instructions from main memory.

Access Time: Much faster than main memory (nanoseconds vs. nanoseconds to nanoseconds).

Function: Acts like a temporary workspace, keeping frequently used information readily available for the processor, reducing access times and boosting performance.

Difference between virtual memory & cache memory:

Virtual Memory:

Focus: Expand the available memory beyond physical RAM limitations.

Size: Much larger than cache memory (gigabytes to terabytes).

Location: No dedicated hardware; utilizes main memory and storage devices (hard disk, SSD).
Content: Allows programs to use more memory than physically available by storing less-used parts on storage devices.
Access Time: Slower than main memory and significantly slower than cache memory (milliseconds to milliseconds vs. nanoseconds).
Function: Creates the illusion of a larger memory space, enabling programs to run smoothly even if their memory requirements exceed physical RAM, improving multitasking and overall system performance.

LTB:
A TLB, or Translation Lookaside Buffer, is a special type of memory cache used in computer systems to speed up memory access. Think of it as a shortcut or cheat sheet for the processor to find frequently used memory locations much faster.
Here's how it works:
Mapping Memory: Every memory location in your computer has a unique address, similar to a house address. These addresses can be long and complex, making them difficult for the processor to work with directly.
Translation and Storage: The TLB acts as a middleman, storing recently used address translations in a small, high-speed cache. These translations map virtual addresses (the addresses used by programs) to their corresponding physical addresses (the actual locations in RAM).
Faster Access: When the processor needs to access data, it first checks the TLB. If the needed address translation is found (a "TLB hit"), the physical address is retrieved instantly and the data can be accessed quickly. This is like checking your address book for a frequently called number instead of dialing the full number every time.
Miss and Fallback: If the address translation is not found in the TLB (a "TLB miss"), the processor must fall back to the slower process of using the page table, a comprehensive list of all address translations stored in main memory. This is like using the phone book when the number isn't in your address book.
Benefits of Using TLB:
Significantly faster memory access: Finding memory locations through the TLB can be many times faster than using the page table, leading to improved program performance and overall system responsiveness.
Reduces processor workload: The TLB frees up the processor from performing frequent page table lookups, allowing it to focus on other tasks.
Improves efficiency: By caching frequently used translations, the TLB minimizes the need for slower accesses to the page table in main memory.
Types of TLBs:
Instruction TLB (ITLB): Specifically stores translations for instruction addresses, crucial for program execution.
Data TLB (DTLB): Stores translations for data addresses used by programs to access various data structures.