COMPUTER ORGANIZATION & ARCHITECTURE

(CSPC 2005)

4TH SEMESTER, COMPUTER SCIENCE & ENGINEERING

MODULE - I

FUNCTIONAL BLOCKS OF A COMPUTER:

A computer consists of five functionally independent main parts:

1. Input
2. Memory
3. Arithmetic and logic
4. Output
5. Control unit

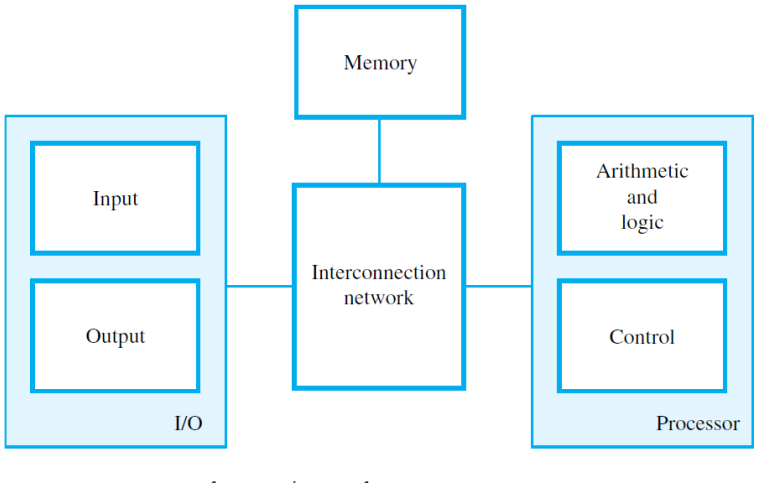


Fig: Basic functional units of a computer

1. The input unit accepts coded information from human operators using devices such as keyboards, or from other computers over digital communication lines.
2. The information received is stored in the computer's memory, either for later use or to be processed immediately by the arithmetic and logic unit.
3. The processing steps are specified by a program that is also stored in the memory.
4. Finally, the results are sent back to the outside world through the output unit. All of these actions are coordinated by the control unit.
5. An interconnection network provides the means for the functional units to exchange information and coordinate their actions.

1. Input Unit:

- Computers accept coded information through input units.
- The most common input device is the keyboard.
- Whenever a key is pressed, the corresponding letter or digit is automatically translated into its corresponding binary code and transmitted to the processor.
- Microphones can be used to capture audio input which is then sampled and converted into digital codes for storage and processing.
- Similarly, cameras can be used to capture video input.

E.g.: Touchpad, mouse, joystick, trackballs, scanners etc are other input devices.

2. Memory Unit:

- The function of the memory unit is to store programs and data.
- There are two classes of storage called **Primary and Secondary**.

Word:

- In computer architecture, a word is a unit of data of a defined bit length that can be addressed and moved between storage and the computer processor.
- The number of bits in each word is referred to as the word length of the computer, typically 16, 32, or 64 bits.
- To provide easy access to any word in the memory, a distinct address is associated with each word location.
- Addresses are consecutive numbers, starting from 0, that identify successive locations.
- A particular word is accessed by specifying its address and issuing a control command to the memory that starts the storage or retrieval process.
- The time required to access one word is called the memory access time.
- This time is independent of the location of the word being accessed.

- **Primary Memory:**
  - Primary memory, also called **main memory**, is a fast memory that operates at electronic speeds.
  - Programs must be stored in this memory while they are being executed.
  - The memory consists of a large number of semiconductor storage cells, each capable of storing one bit of information.
  - The memory is organized so that one word can be stored or retrieved in one basic operation.
  - Instructions and data can be written into or read from the memory under the control of the processor.
  - A memory in which any location can be accessed in a short and fixed amount of time after specifying its address is called a random-access memory (RAM).
- **Cache Memory:**
  - As an adjunct to the main memory, a smaller, faster RAM unit, called a cache, is used to hold sections of a program that are currently being executed, along with any associated data.
  - The cache is tightly coupled with the processor and is usually contained on the same integrated-circuit chip.
  - The purpose of the cache is to facilitate high instruction execution rates.
  - At the start of program execution, the cache is empty.
  - As execution proceeds, instructions are fetched into the processor chip, and a copy of each is placed in the cache.
  - When the execution of an instruction requires data located in the main memory, the data are fetched and copies are also placed in the cache.
- **Secondary Storage:**
  - Secondary memory is used where large amounts of data & programs have to be stored, particularly information that is accessed infrequently.
  - **Examples: -** Magnetic disks & tapes, optical disks (i.e. CD-ROM's), floppies etc.
  - Thus additional, less expensive, permanent secondary storage is used when large amounts of data and many programs have to be stored, particularly for information that is accessed infrequently.
  - Access times for secondary storage are longer than for primary memory.
  - A wide selection of secondary storage devices is available, including magnetic disks, optical disks (DVD and CD), and flash memory devices.

3. **Arithmetic and Logic Unit:**
   - Most computer operations are executed in the arithmetic and logic unit (ALU) of the processor.
   - Any arithmetic or logic operation, such as addition, subtraction, multiplication, division, or comparison of numbers, is initiated by bringing the required operands into the processor, where the operation is performed by the ALU.
   - For example, if two numbers located in the memory are to be added, they are brought into the processor, and the addition is carried out by the ALU.
   - The sum may then be stored in the memory or retained in the processor for immediate use.
   - When operands are brought into the processor, they are stored in high-speed storage elements called registers.
   - Each register can store one word of data.
4. **Output Unit:**
   - The output unit is the counterpart of the input unit.
   - Its function is to send processed results to the outside world.
   - **Examples:** Printer, speakers, monitor etc.
5. **Control Unit:**
   - The memory, arithmetic and logic, and I/O units store and process information and perform input and output operations.
   - The operation of these units must be coordinated in some way.
   - This is the responsibility of the control unit.
   - The control unit is effectively the nerve center that sends control signals to other units and senses their states.
   - Control circuits are responsible for generating the timing signals that govern the transfers and determine when a given action is to take place.
   - In practice, much of the control circuitry is physically distributed throughout the computer.
   - A large set of control lines (wires) carries the signals used for timing and synchronization of events in all units.

Basic Operational Concepts:

- To perform a given task, an appropriate program consisting of a list of instructions is stored in the memory.
- Individual instructions are brought from the memory into the processor, which executes the specified operations.
- Data to be used as instruction operands are also stored in the memory.
- A typical instruction might be

Load R2, LOC

- This instruction reads the contents of a memory location whose address is represented symbolically by the label LOC and loads them into processor register R2.
- The original contents of location LOC are preserved, whereas those of register R2 are overwritten.
- Execution of this instruction requires several steps.

1. First, the instruction is fetched from the memory into the processor.
2. Next, the operation to be performed is determined by the control unit.
3. The operand at LOC is then fetched from the memory into the processor.
4. Finally, the operand is stored in register R2.

Let us consider another example

Add R4, R2, R3

- This instruction adds the contents of registers R2 and R3, then places their sum into register R4.
- The operands in R2 and R3 are not altered, but the previous value in R4 is overwritten by the sum. After completing the desired operations, the results are in processor registers.
- They can be transferred to the memory using instructions such as

Store R4, LOC

- This instruction copies the operand in register R4 to memory location LOC.
- The original contents of location LOC are overwritten, but those of R4 are preserved.
- For Load and Store instructions, transfers between the memory and the processor are initiated by sending the address of the desired memory location and asserting the appropriate control signals.
- The data are then transferred to or from the memory.
- Figure 1.1 shows how the memory and the processor can be connected.

In addition to the ALU and the control circuitry, the processor contains a number of registers used for several different purposes. The instruction register (IR) holds the instruction that is currently being executed. Its output is available to the control circuits, which generate the timing signals that control the various processing elements involved in executing the instruction.

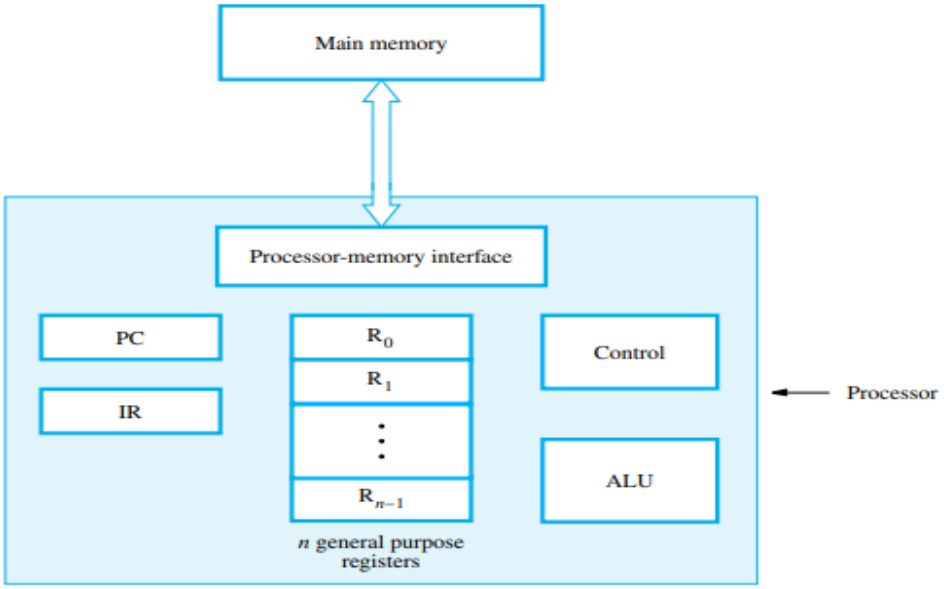


Fig 1.1: ***Connection between the processor and the main memory***

Program Counter (PC):

- The program counter (PC) is another specialized register contains the memory address of the next instruction to be fetched and executed.
- During the execution of an instruction, the contents of the PC are updated to correspond to the address of the next instruction to be executed.

General purpose Registers:

- There are also general-purpose registers R0 through Rn−1, often called processor registers.
- They serve a variety of functions, including holding operands that have been loaded from the memory for processing.

Processor Memory Interface:

- The processor-memory interface is a circuit which manages the transfer of data between the main memory and the processor.
- If a word is to be read from the memory, the interface sends the address of that word to the memory along with a Read control signal.
- The interface waits for the word to be retrieved, then transfers it to the appropriate processor register.
- If a word is to be written into memory, the interface transfers both the address and the word to the memory along with a Write control signal.

Following are typical operating steps:

1) A program must be in the main memory in order for it to be executed. It is often transferred there from secondary storage

2) Execution of the program begins when the PC is set to point to the first instruction of the program.

3) The contents of the PC are transferred to the memory along with a Read control signal. When the addressed word (in this case, the first instruction of the program) has been fetched from the memory it is loaded into register IR. At this point, the instruction is ready to be decoded and executed.

4) If an operand that resides in the memory is required for an instruction, it is fetched by sending its address to the memory and initiating a Read operation. When the operand has been fetched from the memory, it is transferred to a processor register “R”.

5) After operands have been fetched in this way, the ALU can perform a desired arithmetic operation, such as Add, on the values in processor registers. The result is sent to a processor register.

6) If the result is to be written into the memory with a Store instruction, it is transferred from the processor register to the memory, along with the address of the location where the result is to be stored, then a Write operation is initiated.

At some point during the execution of each instruction, the contents of the PC are incremented so that the PC points to the next instruction to be executed. Thus, as soon as the execution of the current instruction is completed, the processor is ready to fetch a new instruction.

Normal execution of a program may be pre-empted if some device requires urgent service. For example, a monitoring device in a computer-controlled industrial process may detect a dangerous condition. In order to respond immediately, execution of the current program must be suspended. To cause this, the device raises an interrupt signal, which is a request for service by the processor. The processor provides the requested service by executing a program called an interrupt-service routine. When the interrupt-service routine is completed, the state of the processor is restored from the memory so that the interrupted program may continue.

Overview of Computer Architecture and Organization:

- **Computer architecture:**
  - Computer architecture is the functional design and structure of a computer system.
  - It focuses on how a computer's hardware components, like the CPU and memory, work together to execute instructions efficiently.
  - It involves decisions about the instruction set, data paths, and control units to optimize performance and functionality.
  - Architecture describes ***what a computer does*** and serves as the blueprint for designing computer systems.
  - It is developed before computer organization during the system design process.

Advantages of Computer Architecture:

- **Performance Optimization:** Proper architectural design is known to advance the efficiency of a system by a large percentage.
- **Flexibility:** This results in the capability to adapt and incorporate new technologies as well as accommodate the different hardware components.
- **Scalability:** Plans should be in a way such that there is provision made for future expansion of a building or accretion.

Disadvantages of Computer Architecture:

- **Complexity:** It may be challenging and huge task which involves time consumption on the flow of design and optimization.
- **Cost:** High-performance architectures need deluxe equipment and parts sometimes that is why they are more expensive.

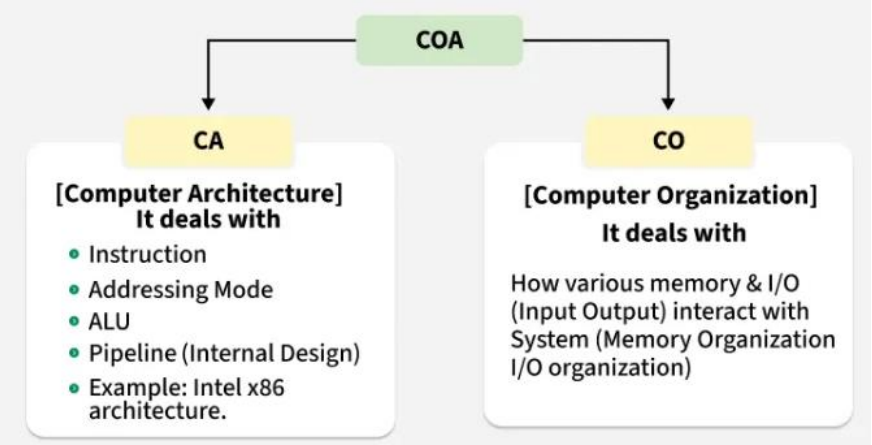
- **Computer Organization:**
  - Computer organization focuses on the physical implementation of a computer system based on its architecture.
  - It deals with how different hardware components, like the CPU, memory, and input/output devices, are connected and work together to execute tasks.
  - While computer architecture explains **what a computer does**, computer organization describes **how it does it**.
  - It focuses on the operational aspects and how hardware components are implemented to support the architecture.
  - Computer organization ensures the architectural design is translated into a functional, physical system.

Advantages of Computer Organization:

- **Practical Implementation:** It offers a perfect account of the physical layout of the computer system.
- **Cost Efficiency:** Organization can help in avoiding wastage of resources hence resulting in a reduction in costs.
- **Reliability:** Organization helps in guaranteeing that similar work produces similar favourable results.

Disadvantages of Computer Organization:

- **Hardware Limitations:** The system's performance is limited by the physical hardware available for installation.
- **Less Flexibility:** The organization however is a lot more well defined and less easy to change once set in its done so.

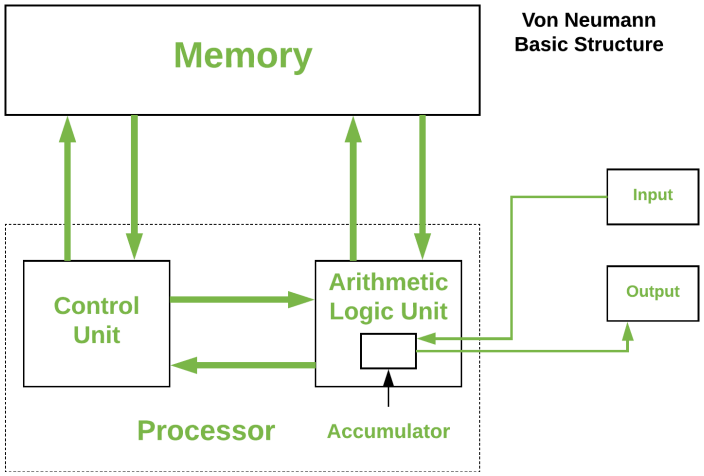


Computer Architecture VS Computer Organization:

Computer Architecture	Computer Organization
Computer Architecture is concerned with the way hardware components are connected together to form a computer system.	Computer Organization is concerned with the structure and behaviour of a computer system as seen by the user.
Architecture describes what the computer does.	The Organization describes how it does it.
Computer Architecture deals with the functional behaviour of computer systems.	Computer Organization deals with a structural relationship.
It acts as the interface between hardware and software.	It deals with the components of a connection in a system.
For designing a computer, its architecture is fixed first.	For designing a computer, an organization is decided after its architecture.
Computer Architecture is also called Instruction Set Architecture (ISA).	Computer Organization is frequently called microarchitecture.
A programmer can view architecture in terms of instructions, addressing modes and registers.	Whereas Organization expresses the realization of architecture.
While designing a computer system architecture is considered first.	An organization is done on the basis of architecture.
Computer Architecture deals with high-level design issues.	Computer Organization deals with low-level design issues.
Architecture involves Logic (Instruction sets, Addressing modes, Data types, Cache optimization)	Organization involves Physical Components (Circuit design, Adders, Signals, Peripherals)
The different architectural categories found in our computer systems are as follows: • Von-Neumann Architecture • Harvard Architecture • Instruction Set Architecture • Micro-architecture • System Design	CPU organization is classified into three categories based on the number of address fields: • Organization of a single Accumulator. • Organization of general registers • Stack organization

Von Neumann Architecture:

- The Von-Neumann Architecture or Von-Neumann model is also known as **"Princeton Architecture"**.
- This architecture was published by the Mathematician **John Von Neumann** in **1945**.
- This architecture implemented the stored program concept in which the data and instructions are stored in the same memory.
- This architecture consists of a CPU (ALU, Registers, Control Unit), Memory and I/O unit.



Following are the components of Von Neumann Architecture:

1. CPU (Central processing unit)
    - CU (Control Unit)
    - ALU (Arithmetic and logic unit)
    - Registers
        - PC (Program Counter)
        - IR (Instruction Register)
        - AC (Accumulator)
        - MAR (Memory Address Register)
        - MDR (Memory Data Register)
2. BUSES
3. I/o Devices
4. Memory Unit

1. CPU: CPU acts as the brain of the computer and is responsible for the execution of instructions.

a) Control Unit: A control unit (CU) handles all processor control signals. It directs all input and output flow, fetches code for instructions, and controls how data moves around the system.

b) Arithmetic and Logic Unit (ALU): The arithmetic logic unit is that part of the CPU that handles all the calculations the CPU may need, e.g. Addition, Subtraction, Comparisons. It performs Logical Operations, Bit Shifting Operations, and Arithmetic operations.

c) Registers: A processor based on von Neumann architecture has five special registers which it uses for processing:

- **Program counter (PC)** holds the memory address of the next instruction to be fetched from primary storage.
- **Memory Address Register (MAR)** holds the address of the current instruction that is to be fetched from memory, or the address in memory to which data is to be transferred.
- **Memory Data Register (MDR)** holds the contents found at the address held in MAR or data which is to be transferred to the primary storage.
- **Current Instruction Register (CIR)** holds the instruction that is currently being decoded and executed.
- **Accumulator:** It is a special purpose Register that stores intermediate results of arithmetic and logic operations.

2. BUSES:

- The bus is a communication system that transfers data, addresses, and control signals between the CPU, memory, and I/O devices.
- Group of lines that serve as connecting path for several devices is called a bus (one bit per line).
- In Von Neumann architecture, a single bus is shared for both data and instructions, which can create a bottleneck known as the Von Neumann bottleneck.
- There are three types of BUSES
    a) **Data Bus:** It carries data among the memory unit, the I/O devices, and the processor.
    b) **Address Bus:** It carries the address of data (not the actual data) between memory and processor.
    c) **Control Bus:** It carries control commands from the CPU (and status signals from other devices) in order to control and coordinate all the activities within the computer.

3. I/o Devices:

- Program or data is read into main memory from the *input device* or secondary storage under the control of CPU input instruction.
- *Output devices* are used to output the information from a computer.
- If some results are evaluated by CPU and it is stored in the computer, then with the help of output devices, we can present them to the user.

4. Memory:

- A memory unit is a collection of storage cells together with associated circuits needed to transfer information in and out of the storage.
- The memory stores binary information in groups of bits called words. The internal structure of a memory unit is specified by the number of words it contains and the number of bits in each word.

There are two types of Primary Memory:

1) **RAM:** VOLATILE MEMORY or temporary Memory **(to store the program in execution)**
2) **ROM:** NON-VOLATILE MEMORY or permanent Memory **(to store the booting program)**

- **Key Characteristics of Von Neumann Architecture:**

1. **Single Memory for Data and Instructions:** Both data and program instructions are stored in the same memory.
2. **Shared Bus:** A single bus is used for transferring data, addresses, and control signals, which can limit performance.
3. **Sequential Execution:** Instructions are executed one at a time in a sequential manner.

- **Advantages of Von Neumann Architecture:**
    - **Simplified Design**: Uses a single memory for data and instructions, reducing hardware complexity.
    - **Cost-Effective**: Lower production costs due to fewer components.
    - **Flexibility**: Can run various programs and makes it suitable for general-purpose computing.
    - **Ease of Programming**: Unified memory structure simplifies software development.
    - **Widely Adopted**: Forms the foundation of most modern computers hence, ensures widespread compatibility.
- **Limitations of Von Neumann Architecture:**
    - **Memory Bottleneck**: Shared memory slows down data and instruction transfer.
    - **Sequential Processing**: Cannot process data and instructions simultaneously.
    - **Scalability Issues**: Struggles with high-performance tasks requiring rapid memory access.
    - **Energy Inefficiency**: Frequent memory access increases power consumption.
    - **Latency**: Data and instruction fetch delays reduce overall system efficiency.
- **Applications of Von Neumann Architecture:**
    - **General-Purpose Computing**: Powers desktops, laptops, and smartphones.
    - **Embedded Systems**: Used in simple devices where cost and simplicity are priorities.
    - **Software Development**: Shapes programming tools and languages due to its unified structure.
    - **Education**: A foundational concept in computer science courses.
    - **Gaming and Multimedia**: Supports complex applications like video games and editing software.

Instruction Set Architecture (ISA):

- It is a collection of machine language instruction that a particular processor understands and execute.in other words set of assembly language mnemonics represents machine code of a particular computer.
- The *Instruction Set Architecture* (ISA) is the part of the processor that is visible to the programmer or compiler writer.
- The ISA serves as the boundary between software and hardware.
- The ISA of a processor can be described using 5 categories:
    1. **Operand Storage in the CPU:** Where are the operands kept other than in memory?
    2. **Number of explicit named operands:** How many operands are named in a typical instruction.
    3. **Operand location:** Can any ALU instruction operand be located in memory? Or must all operands be kept internally in the CPU?
    4. **Operations:** What operations are provided in the ISA?
    5. **Type and size of operands:** What is the type and size of each operand and how is it specified?

The 3 most common types of ISAs are:

1. ***Stack*** - The operands are implicitly on top of the stack.
    - **Advantages:** Simple Model of expression evaluation (reverse polish). Short instructions.
    - **Disadvantages:** A stack can't be randomly accessed This makes it hard to generate efficient code. The stack itself is accessed every operation and becomes a bottleneck.

2. ***Accumulator*** - One operand is implicitly the accumulator.
    - **Advantages:** Short instructions.
    - **Disadvantages:** The accumulator is only temporary storage so memory traffic is the highest for this approach.

3. ***General Purpose Register (GPR)*** - All operands are explicitly mentioned; they are either registers or memory locations.
    - **Advantages:** Makes code generation easy. Data can be stored for long periods in registers.
    - **Disadvantages:** All operands must be named leading to longer instructions.

Let's look at the assembly code of

C = A + B;

in all 3 architectures:

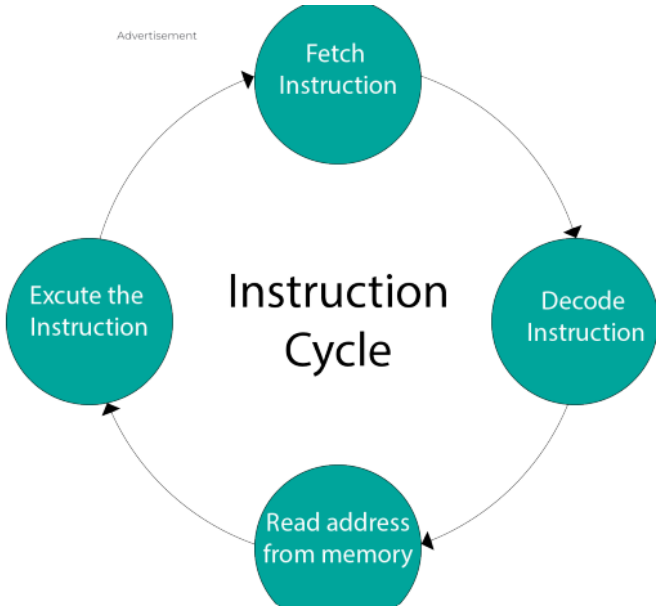
STACK	ACCUMULATOR	GENERAL PURPOSE REGISTER
PUSH A	LOAD A	LOAD R1, A
PUSH B	ADD B	ADD R1, B
ADD	STORE C	STORE R1, C
POP C	-	-

Not all processors can be neatly tagged into one of the above categories.

Instruction Cycle:

A program residing in the memory unit of a computer consists of a sequence of instructions. These instructions are executed by the processor by going through a cycle for each instruction.
In a basic computer, each instruction cycle consists of the following phases:

1. Fetch instruction from memory.
2. Decode the instruction.
3. Read the effective address from memory.
4. Execute the instruction.



The Instruction Execution Cycle of the CPU:

- The **CPU** is a **machine** which perform the task i.e. execution of a computer instruction.
    - **Like** any **machine**, a **CPU** will **repeatedly** perform a (fixed) **number of "steps"** or "**phases"**
    - The **steps/phases** that a **CPU** go through will **execute the *next* instruction** in the **computer program**
  - The **steps** that are performed by the **CPU** are **collectively** called the **CPU's instruction execution cycle.**

- The **Instruction Execution Cycle** of the **CPU** consists of the following **4 steps**:
  1. The **instruction fetch** step/phase:
     - During this phase, the **CPU** will **fetch** the ***next* instruction** from memory and store it in the **Instruction Register (IR).**
  2. The **instruction decode** step/phase:
     - During this phase, the **CPU** decode the **instruction (a binary number!)** in the **Instruction Register (IR)** and determine all the **operands** that it needs to **execute** the instruction.
  3. The **operand fetch** step/phase:
     - During this phase, the **CPU** fetches the **operands** from the **registers** to the **inputs** of the **Arithmetic/Logic Unit (ALU)**
  4. The **instruction execute** step/phase:
     - In this phase, the **CPU** performs the **arithmetic/logic operation** and **save** the **result** in some **register**

A detailed look at the instruction execution cycle:

1. The **instruction fetch** step:
    - In this step, the **CPU** will **fetch** the **instruction** from the **memory** that is stored at the **memory address** given by the value inside the **Program Counter (PC)** into the **Instruction Register** of the **CPU.**
    - Recall that the **Program Counter (special register)** contains the **address (= location in memory)** of the ***next*** **instruction**.
    - So, in this step, the **CPU** will **fetch a copy the *next* instruction** from the **memory** into its **Instruction register (IR)**.
    - After fetching the instruction from memory, the **CPU** will **update (= increase)** the **Program Counter** so it will contain the **address** of the ***subsequent*** **instruction**.

➢ When this step is completed:

I. The Instruction Register inside the CPU will contain the next instruction that the CPU must execute
    - The CPU prepared itself to fetch a ***new*** **next instruction**
        - So, this preparation will ingeniously enable the **repetition** of the **same steps** to fetch and **execute** a ***different*** **instruction**.

2. The **instruction decode** step:
    - The **instruction** is represented by a **binary pattern** because the computer memory can only store binary patterns.
    - Each **binary pattern** will **represent** a certain **computer operation.**
    - Before the CPU can **perform an instruction**, the CPU must **find out** what is the **instruction** that is represented (= **encoded**) by the **binary pattern.**
    - The CPU contains **circuitry** to **decode** the instruction's binary pattern.
        i. What **operation** the CPU needs to perform (e.g.: add, subtract, multiply, divide, negate, logical AND, logical OR, logical NOT, etc.)
        ii. Which **operands** the CPU needs to use in the **operation** and where to **store** the **result** (e.g.: if the instruction was "add", then which input values should "add" use, and where to store the sum)
3. The **operand fetch** step:
    - After **identifying** the **input operands** in the **instruction decode** step, the **operand fetch** step will **obtain (= fetch)** the **input values (= operands)** for the **instruction.**
    - The **values** are transported **electronically** through **wires** inside the **CPU** to the **Arithmetic/Logic Unit (ALU)** where the **operation** take place.

- When this step is **complete**, the **CPU** will:
    - Have **determined** the **operation** to perform
    - Obtained the **operands** needed to perform the **operation**

4. The **instruction executing** step:
    - In this **final step**, the **CPU** will perform the **operation** specified by the **instruction code** on the **fetched operands**
    - **After** performing the **operation**, the ***result*** of the **operation** will be **saved** in the **designated register**

After the **instruction execution** step, the **operation** specified in the **instruction** that was fetched in **step 1** of the **Instruction Execution Cycle** is **complete**.

The **CPU** can **discard** the **instruction** in the **Instruction Register.**

(Don't need to keep the instruction or save it back in memory, because the instruction in the **Instruction Register** is a **copy** transferred from memory.

So just like we can **discard** a **meal order** at **McDonalds** or some **restaurant after fulfilling** the order,
the **CPU** can **discard** the instruction after executing (= "fulfilling") the operation.

Notice that:

- The **CPU** has **prepared** itself to **fetch** a ***new*** **"next instruction"** (= the instruction that follows the fetched instruction) in **Step 1** of the **Instruction Execution Cycle.**

The **CPU** can now **repeat** the **Instruction Execution Cycle** again.

When the CPU repeats the **Instruction Execution Cycle**, it will fetch a ***different*** **instruction** because the CPU has made the necessary preparation in **Step 1**.

Micro-operations:

- The operations performed on the data stored in registers are called **Micro-operations**.
- A microoperation is an elementary operation performed on the information stored in one or more registers.
- The result of the operation may replace the previous binary information of a register or may be transferred to another register.
- Examples of microoperations are shift, count, clear, and load.

Register Transfer Language:

- The Register Transfer Language is the symbolic representation of notations used to specify the sequence of micro-operations.
- Register Transfer Language **(RTL)** is a low-level language that is used to describe the functioning of a digital circuit and the transfer of information between registers.
- It provides how data moves from one register to the other and how data is processed within the digital system.
- Through RTL, there is a capability of creating abstraction levels where high-level design descriptions can be created and easily linked to low-level hardware implementation in designing, simulating, as well as synthesizing digital circuits.

Components of RTL:

The components of Register Transfer Language are, -

- **Registers:** These are storage elements that store data. Think of them as small and swift storage spaces located within the CPU.
- **Operations:** The operations are carried out on data stored in registers. It includes activities like addition, subtraction, and logic.
- **Control Signals:** These signals control and manage the system's functions and data handling.

These components assist in comprehending the flow of data and how it is manipulated within the computer.

Examples of Simple Register Transfers:

Here are some basic examples of register transfer language e for better understanding.

1. **Load Operations:** This is a process of transferring data from memory to a register.
    - Example: M [100] -> R1 (Transfer the value stored at memory location 100 into register R1).
2. **Move Operations:** This entails moving data from one register to the other.
    - Example: T3 <- T2 (Copy the content of register T2 into T3).

These examples show the basic movements and operations within a CPU using **Register Transfer Language**.

Common Symbols and Notation Used in RTL:

In RTL, different symbols are assigned to various operations and components. Therefore, it is important to gain knowledge and comprehend these symbols to comprehend and write in RTL.

- *R1, R2, R3*… represent registers.
- *M [ ]* represents memory locations.
- <- denotes data transfer.

Register Transfer:

- Computer registers are denoted by capital letters (sometimes followed by numerals) to denote the function of the register.
- The register that holds an address for the memory unit is usually called a **memory address register** and is denoted by **MAR**.
- Other registers are PC (for program counter), IR (for instruction register, and R1 (for processor register).
- An n-bit register is sequence of n-flipflops numbered from 0 through n-1, starting from 0 in the rightmost position and increasing the numbers toward the left.
- The most common way to represent a register is by a rectangular box with the name of the register inside, as shown in the figure below.
- The individual bits can be distinguished as shown in (b).
- The numbering of bits in a 16-bit register can be marked on top of the box as shown in (c).
- A16-bit register is partitioned into two parts in (d).
- Bits 0 through 7 are assigned the symbol L (for low byte) and bits 8 through 15 are assigned the symbol H (for high byte). The name of the 16-bit register is PC. The symbol PC (0-7) or PC (L) refers to the low-order byte and PC (8-15) or PC (H) to the high-order byte.

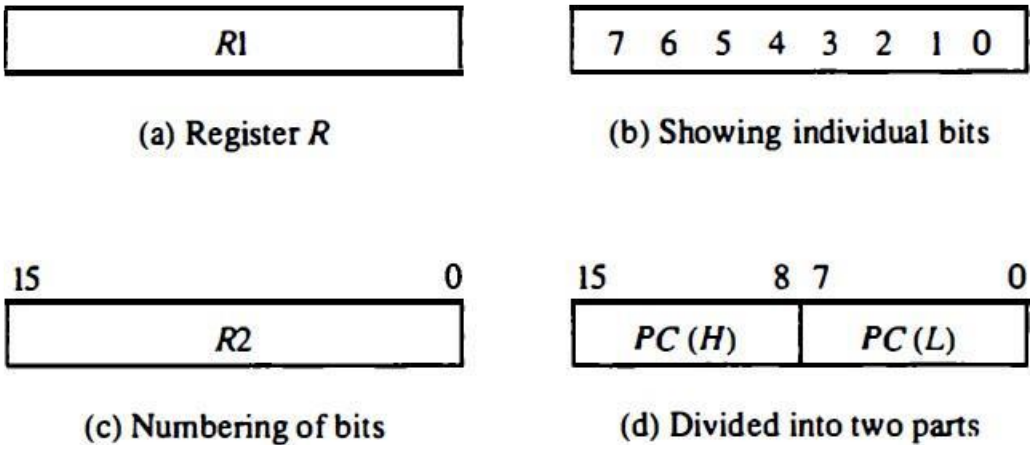


Fig: Block diagram of registers

- Information transfer from one register to another is designated in symbolic form by means of a replacement operator as shown below, which denotes a transfer of the contents of register R1 into register R2.
- Contents of R2 are replaced by the contents of R1.
- By definition, the content of the source register R1 does not change after the transfer.
- Register transfer implies that circuits are available from the outputs of the source register to the inputs of the destination register.

R2 <--R1

- Sometimes, we may want the transfer to occur only under a predetermined control condition.
- This can be shown by means of an if-then statement

If (P = 1) then (R2 <--R1)

- where P is a control signal generated in the control section. A control function is a Boolean variable that is equal to 1 or 0. The control function is included in the statement as follows:

P: R2 <--R1

- The control condition is terminated with a colon. It symbolizes the requirement that the transfer operation be executed by the hardware only if P = 1.
- Every statement written in a register transfer notation implies a hardware construction for implementing the transfer.
- Registers are denoted by capital letters and numerals may follow the letters.
- Parentheses are used to denote a part of a register by specifying the range of bits or by giving a symbol name to a portion of a register.
- The arrow denotes a transfer of information and the direction of transfer.
- A comma is used to separate two or more operations that are executed at the same time.
- The statement

T: R2 <- R1, R1 <- R2

- It denotes an operation that exchanges the contents of two registers during one common clock pulse provided that T = 1.
- The basic symbols of the register transfer notation are given below:

Symbol	Description	Examples
Letters (and numerals)	Denotes a register	*MAR*, *R2*
Parentheses ( )	Denotes a part of a register	*R2*(0–7), *R2*(*L*)
Arrow ←	Denotes transfer of information	*R2* ← *R1*
Comma ,	Separates two microoperations	*R2* ← *R1*, *R1* ← *R2*

Fig: Basic symbols of register Transfer

Advantages of Register Transfer Language (RTL)

- Enables efficient hardware design.
- This makes it possible to simulate some activities and perhaps detect some errors at an early date.
- Implements conceptual description up to the gate-level hardware.
- It helps to reuse the design components.
- It gives a clear guide on how to do timing analysis on a given design.

Disadvantages of Register Transfer Language (RTL)

- Although the performance is high, this type can be quite challenging to debug.
- These may lead to inefficient large constructions if not well optimized for beneficial use.
- The reasoning behind it is rather hardware-oriented and may cause problems that are hard to comprehend without knowledge of message flow.
- Synthesis results appear to depend on the capabilities of specific tools.
- Compared to high-level descriptions, the amount of abstraction retrieved by this system is very low.

Addressing Modes-

- The different ways of specifying the location of an operand in an instruction are called as **addressing modes**.
- The term addressing modes refers to the way in which the operand of an instruction is specified.
- The addressing mode specifies a rule for interpreting or modifying the address field of the instruction before the operand is actually executed.

Types of Addressing Modes-

In computer architecture, there are following types of addressing modes-

1. Implied / Implicit Addressing Mode
2. Stack Addressing Mode
3. Immediate Addressing Mode
4. Direct Addressing Mode
5. Indirect Addressing Mode
6. Register Direct Addressing Mode
7. Register Indirect Addressing Mode
8. Relative Addressing Mode
9. Indexed Addressing Mode
10. Base Register Addressing Mode
11. Auto-Increment Addressing Mode
12. Auto-Decrement Addressing Mode

1. Implied Addressing Mode:

- In this addressing mode, the definition of the instruction itself specifies the operands implicitly. It is also called as implicit addressing mode.

  e.g. • The instruction "Complement Accumulator" is an implied mode instruction (CMA).

  • In a stack organized computer, zero address instructions are implied mode instructions.

2. Stack Addressing Mode:

- In this addressing mode, the operand is contained at the top of the stack.

  e.g. ADD

  - This instruction simply pops out two symbols contained at the top of the stack.
  - The addition of those two operands is performed.
  - The result so obtained after addition is pushed again at the top of the stack.

3. Immediate Addressing Mode:

- In this addressing mode, the operand is specified in the instruction explicitly. Instead of address field, an operand field is present that contains the operand.

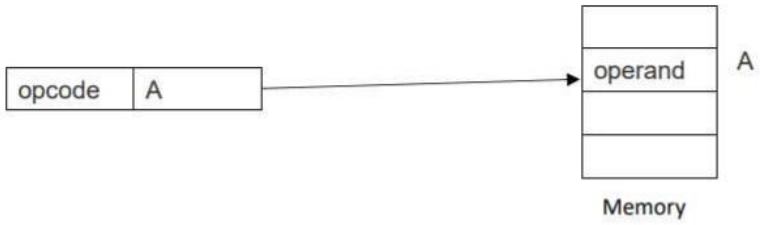
Immediate Addressing Mode

e.g.

- ADD 100 will increment the value stored in the accumulator by 10.
- MOV R #20 initialized register R to a constant value 20.

4. Direct Addressing Mode:

- In this addressing mode, the address field of the instruction contains the effective address of the operand. Only one reference to memory is required to fetch the operand. It is also called as **absolute addressing mode**.



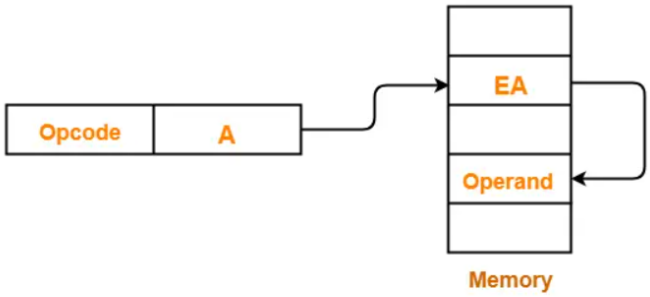
Example-

ADD X will increment the value stored in the accumulator by the value stored at memory location X.

AC ← AC + [X]

5. Indirect Addressing Mode:

- In this addressing mode, the address field of the instruction specifies the address of memory location that contains the effective address of the operand. Two references to memory are required to fetch the operand.



Indirect Addressing Mode

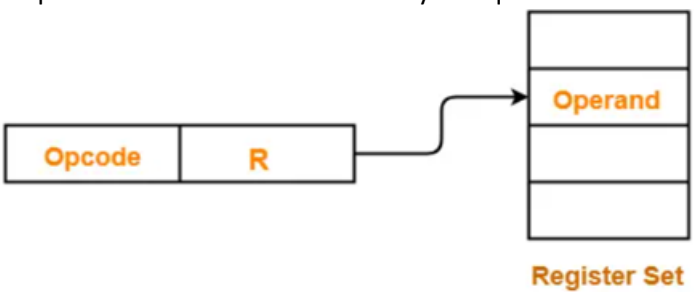
Example-

ADD X will increment the value stored in the accumulator by the value stored at memory location specified by X.

AC ← AC + [[X]]

6. Register Direct Addressing Mode:

- In this addressing mode, the operand is contained in a register set. The address field of the instruction refers to a CPU register that contains the operand. No reference to memory is required to fetch the operand.



Example-

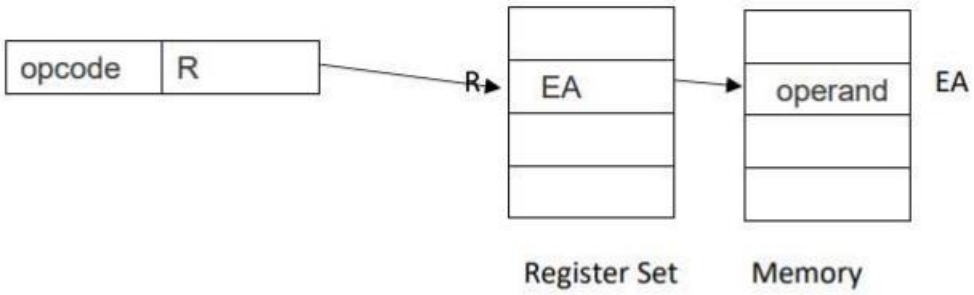
ADD R will increment the value stored in the accumulator by the content of register R.

AC ← AC + [R]

- This addressing mode is similar to direct addressing mode.
- The only difference is address field of the instruction refers to a CPU register instead of main memory.

7. Register Indirect Addressing Mode:

- In this addressing mode, the address field of the instruction refers to a CPU register that contains the effective address of the operand. Only one reference to memory is required to fetch the operand.



Example-

ADD R will increment the value stored in the accumulator by the content of memory location specified in register R.

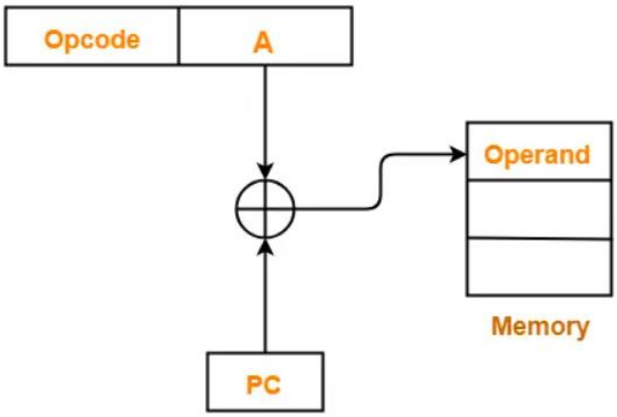
AC ← AC + [[R]]

- This addressing mode is similar to indirect addressing mode.
- The only difference is address field of the instruction refers to a CPU register.

8. Relative Addressing Mode-

In this addressing mode, Effective address of the operand is obtained by adding the content of program counter with the address part of the instruction.

Effective Address = Content of Program Counter + Address part of the instruction

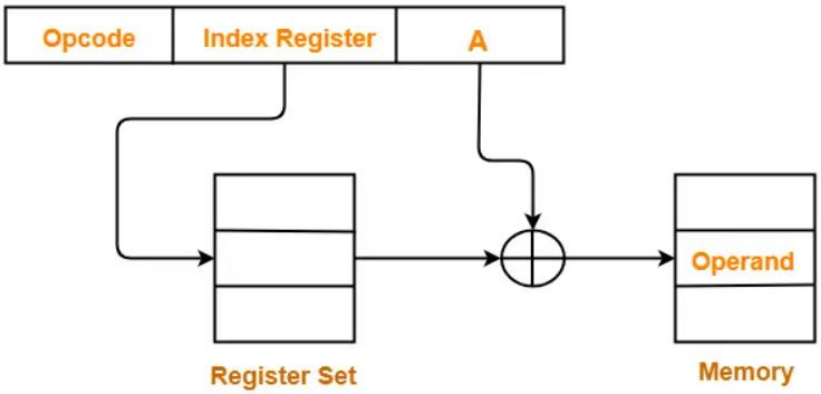


- **Program counter** (PC) always contains the address of the next instruction to be executed.
- After fetching the address of the instruction, the value of program counter immediately increases.
- The value increases irrespective of whether the fetched instruction has completely executed or not.

9. Indexed Addressing Mode-

In this addressing mode, Effective address of the operand is obtained by adding the content of index register with the address part of the instruction.

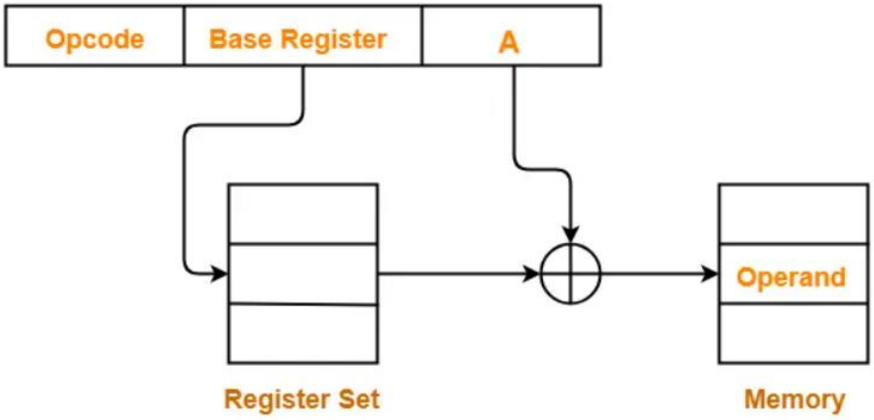
Effective Address = Content of Index Register + Address part of the instruction



10. Base Register Addressing Mode-

In this addressing mode, Effective address of the operand is obtained by adding the content of base register with the address part of the instruction.

Effective Address = Content of Base Register + Address part of the instruction



11. Auto-Increment Addressing Mode-

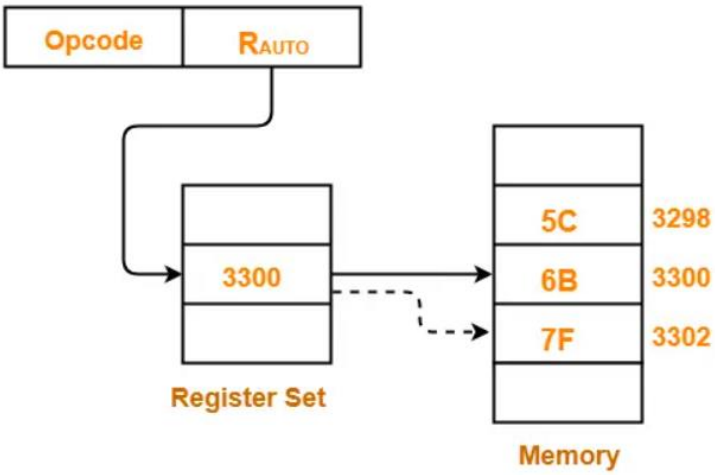
This addressing mode is a special case of Register Indirect Addressing Mode where-

Effective Address of the Operand = Content of Register

In this addressing mode,

- After accessing the operand, the content of the register is automatically incremented by step size 'd'.
- Step size 'd' depends on the size of operand accessed.
- Only one reference to memory is required to fetch the operand.

Example-



Assume operand size = 2 bytes.

Here,

- After fetching the operand 6B, the instruction register R_{AUTO} will be automatically incremented by 2.
- Then, updated value of R_{AUTO} will be 3300 + 2 = 3302.
- At memory address 3302, the next operand will be found.

NOTE-

In auto-increment addressing mode,

- First, the operand value is fetched.
- Then, the instruction register R_{AUTO} value is incremented by step size 'd'.

12. Auto-Decrement Addressing Mode-

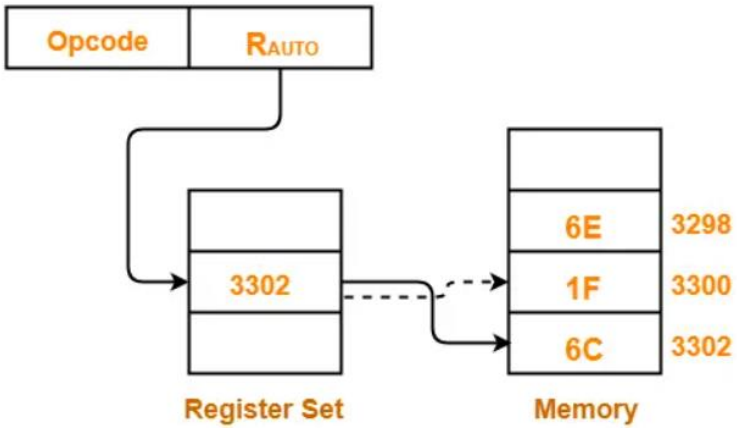
This addressing mode is again a special case of Register Indirect Addressing Mode where-

Effective Address of the Operand = Content of Register – Step Size

In this addressing mode,

- First, the content of the register is decremented by step size 'd'.
- Step size 'd' depends on the size of operand accessed.
- After decrementing, the operand is read.
- Only one reference to memory is required to fetch the operand.

Example-



Assume operand size = 2 bytes.

Here,

- First, the instruction register R_{AUTO} will be decremented by 2.
- Then, updated value of R_{AUTO} will be 3302 – 2 = 3300.
- At memory address 3300, the operand will be found.

NOTE-

In auto-decrement addressing mode,

- First, the instruction register R_{AUTO} value is decremented by step size 'd'.
- Then, the operand value is fetched.

Instruction set:

- A set of codes that can only be understood by a processor of the computer or CPU is known as an **instruction set**.
- These codes and machine languages are generally present as 1s and 0s. The movements of bits and bytes are controlled by these instruction sets present in the processor.
- Some common examples of instruction sets are:
    1. JUMP – jump instruction set is used to jump to any designated address of RAM.
    2. ADD – add instruction set is used to add any two numbers together.
    3. LOAD – load instruction set is used to load any required information from the RAM to the CPU.

Types of instruction set:

Most computer instructions can be classified into three categories:

1. Data transfer instructions
2. Data manipulation instructions
3. Program control instructions

1) Data Transfer Instructions:

- Data transfer instructions move data from one place in the computer to another without changing the data content.
- The most common transfers are between memory and processor registers, between processor registers and input or output, and between the processor registers themselves.
- Table below gives a list of eight data transfer instructions used in many computers.

Name	Mnemonic
Load	LD
Store	ST
Move	MOV
Exchange	XCH
Input	IN
Output	OUT
Push	PUSH
Pop	POP

- Accompanying each instruction is a mnemonic symbol. Different computers use different mnemonics for the same instruction name.
- The **load instruction** has been used mostly to designate a transfer from memory to a processor register, usually an accumulator.
- The **store instruction** designates a transfer from a processor register into memory.
- The **move instruction** has been used in computers with multiple CPU registers to designate a transfer from one register to another.
- It has also been used for data transfers between CPU registers and memory or between two memory words.
- The **exchange instruction** swaps information between two registers or a register and a memory word.
- The **input and output instructions** transfer data among processor registers and input or output terminals.
- The **push and pop** instructions transfer data between processor registers and a memory stack.

2) Data Manipulation Instructions:

The data manipulation instructions in a typical computer are usually divided into three basic types:

a) Arithmetic instructions

b) Logical and bit manipulation instructions

c) Shift instructions

a) Arithmetic instructions:

- The four basic arithmetic operations are addition, subtraction, multiplication and division.
- Most computers provide instructions for all four operations.
- Some small computers have only addition and possibly subtraction instructions.
- A list of typical arithmetic instructions is given in Table given below:

Name	Mnemonic
Increment	INC
Decrement	DEC
Add	ADD
Subtract	SUB
Multiply	MUL
Divide	DIV
Add with carry	ADDC
Subtract with borrow	SUBB
Negate (2's complement)	NEG

- The **increment instruction** adds 1 to the value stored in a register or memory word.
- The **decrement instruction** subtracts 1 from a value stored in a register or memory word.
- The add, subtract, multiply, and divide instructions may be available for different types of data.
- The data type assumed to be in processor registers during the execution of these arithmetic operations is included in the definition of the operation code.
- An arithmetic instruction may specify fixed-point or floating-point data, binary or decimal data, single-precision or double-precision data.

The mnemonics for three add instructions that specify different data types are shown below:

- **ADDI** Add two binary integer numbers **ADDF** Add two floating-point numbers **ADDD** Add two decimal numbers in BCD
- The instruction "**add with carry**" performs the addition on two operands plus the value of the carry from the previous computation.
- Similarly, the "**subtract with borrow**" instruction subtracts two words and a borrow which may have resulted from a previous subtract operation.
- The **negate instruction** forms the 2' s complement of a number, effectively reversing the sign of an integer when represented in the signed- 2's complement form.

b) Logical and Bit Manipulation Instructions:

- Logical instructions **perform binary operations on strings of bits** stored in registers.
- They are useful for manipulating individual bits or a group of bits that represent binary-coded information.
- The **logical instructions consider each bit of the operand separately** and treat it as a Boolean variable.
- By proper application of the logical instructions, it is possible to change bit values, to clear a group of bits, or to insert new bit values into operands stored in registers or memory words.
- Some logical and bit manipulation instructions are shown in the figure below:

Name	Mnemonic
Clear	CLR
Complement	COM
AND	AND
OR	OR
Exclusive-OR	XOR
Clear carry	CLRC
Set carry	SETC
Complement carry	COMC
Enable interrupt	EI
Disable interrupt	DI

- The clear instruction causes the specified operand to be replaced by D's.
- The complement instruction produces the 1's complement by inverting all the bits of the operand.
- The AND, OR, and XOR instructions produce the corresponding logical operations on individual bits of the operands.
- Although they perform Boolean operations, when used in computer instructions, the logical instructions should be considered as performing bit manipulation operations.
- There are three-bit manipulation operations possible: a selected bit can be cleared to 0, or can be set to 1, or can be complemented.
- The three logical instructions are usually applied to do just that.

c) Shift Instructions:

- Shifts are operations in which the bits of a word are moved to the left or right.
- Shift instructions may specify either logical shifts, arithmetic shifts, or rotate-type operations.
- In either case the shift maybe to the right or to the left.
- Table below lists four types of shift instructions:

Name	Mnemonic
Logical shift right	SHR
Logical shift left	SHL
Arithmetic shift right	SHRA
Arithmetic shift left	SHLA
Rotate right	ROR
Rotate left	ROL
Rotate right through carry	RORC
Rotate left through carry	ROLC

- The **logical shift** inserts 0to the end bit position.
- The end position is the leftmost bit for shift right and the rightmost bit position for the shift left.

- The **arithmetic shift-right** instruction must preserve the sign bit in the leftmost position.
- The sign bit is shifted to the right together with the rest of the number, but the sign bit itself remains unchanged.
- This is a shift-right operation with the end bit remaining the same.
- The arithmetic shift-left instruction inserts 0 to the end position and is identical to the logical shift-left instruction.
- The rotate instructions produce a circular shift.
- Bits shifted out at one end of the word are not lost as in a logical shift but are circulated back into the other end.
- The rotate through carry instruction treats a carry bit as an extension of the register whose word is being rotated.
- Thus, a rotate-left through carry instruction transfers the carry bit into the rightmost bit position of the register, transfers the leftmost bit position into the carry, and at the same time, shifts the entire register to the left.

3. Program Control Instructions:

- Program control instructions provide decision-making capabilities and change the path taken by the program when executed in the computer a program control type of instruction, when executed, may change the address value in the program counter and cause the flow of control to be altered.
- In other words, program control instructions specify conditions for altering the content of the program counter.
- Some program control instructions are listed in Table below:

Name	Mnemonic
Branch	BR
Jump	JMP
Skip	SKP
Call	CALL
Return	RET
Compare (by subtraction)	CMP
Test (by ANDing)	TST

- Branch and jump instructions are used interchangeably to mean the same thing, but sometimes they are used to denote different addressing modes.
- Branch instruction is written as **BR ADR**, where ADR is a symbolic name for an address.
- Branch and jump instructions may be conditional or unconditional.
- An unconditional branch instruction causes a branch to the specified address without any conditions.
- The conditional branch instruction specifies a condition such as branch if positive or branch if zero.
- If the condition is met, the program counter is loaded with the branch address and the next instruction is taken from this address.
- If the condition is not met, the program counter is not changed and the next instruction is taken from the next location in sequence.
- The **skip instruction** does not need an address field and is therefore a zero-address instruction.
- A conditional skip instruction will skip the next instruction if the condition is met.
- If the condition is not met, control proceeds with the next instruction in sequence.
- The **call and return** instructions are used in conjunction with subroutines.
- The **compare instruction** performs a subtraction between two operands, but the result of the operation is not retained.
- However, certain status bit conditions are set as a result of the operation.
- Similarly, the **test instruction** performs the logical AND of two operands and updates certain status bits without retaining the result or changing the operands.