DH SIR'S CLASSROOM
SUBJECT: Computer Organisation
COURSE: B.TECH IN CSE[3rd sem]
Module – 1:

………..

………..

Contents:

Basic organization of the stored program computer and operation sequence for execution of a program. Role of operating systems and compiler/assembler. Fetch, decode and execute cycle, Concept of operator, operand, registers and storage, Instruction format. Instruction sets and addressing modes.

Commonly used number systems. Fixed and floating point representation of numbers.

………..

………..

Basic organization of the stored program computer and operation sequence for execution of a program:

The basic organization of a stored program computer involves the architecture and components that allow it to store and execute programs. Here is an overview of the key elements and the operation sequence for the execution of a program:

1. **Central Processing Unit (CPU):**
   - The CPU is the brain of the computer and is responsible for executing instructions.
   - It typically consists of an Arithmetic Logic Unit (ALU) for mathematical and logical operations and a Control Unit (CU) for instruction interpretation and execution.
2. **Memory:**
   - Memory is used to store both data and instructions that the CPU will execute.
   - The two main types of memory are:
     - **RAM (Random Access Memory):** Volatile memory used for temporary storage during program execution.
     - **ROM (Read-Only Memory):** Non-volatile memory that stores the computer's firmware or permanent instructions.
3. **Input/Output (I/O) Devices:**
   - These devices allow the computer to communicate with the external world.
   - Examples include keyboards, monitors, printers, and storage devices.
4. **Control Bus, Address Bus, and Data Bus:**
   - These buses facilitate communication between different components of the computer.
   - The **Control Bus** carries control signals (e.g., read, write, reset) between the CPU and other parts of the computer.
   - The **Address Bus** carries the address of the memory location to be accessed.
   - The **Data Bus** carries the actual data between the CPU and memory or I/O devices.

Operation Sequence for Execution of a Program:

1. **Fetch:**
   - The CPU fetches the next instruction from memory. The address of the instruction is provided by the program counter (PC).
2. **Decode:**

- The fetched instruction is decoded by the Control Unit to determine the operation to be performed.

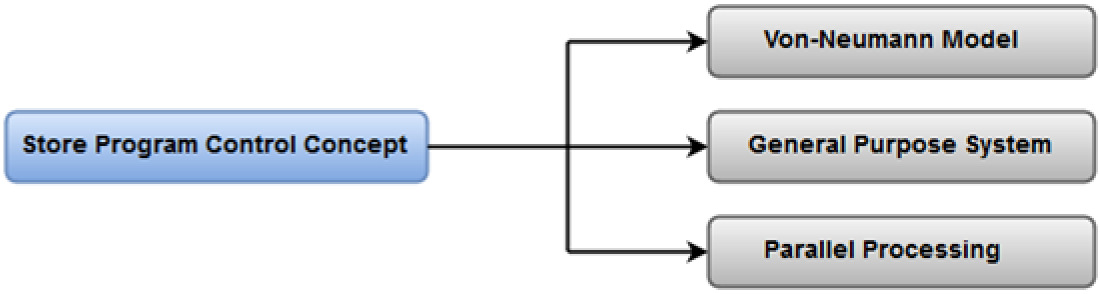
3. **Execute:**
    - The CPU performs the operation specified by the decoded instruction using the ALU and other relevant components.
4. **Store (Optional):**
    - If the operation involves storing data back to memory, the result is stored in the specified memory location.
5. **Update Program Counter:**
    - The program counter is updated to point to the next instruction in memory, allowing the CPU to fetch the next instruction in the sequence.
6. **Repeat:**
    - Steps 1-5 are repeated until the program is complete.

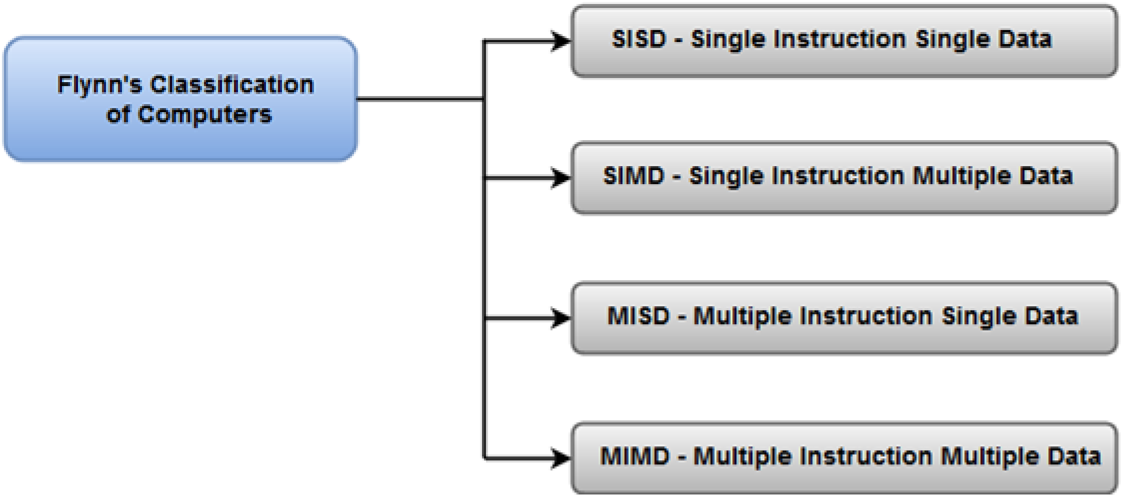
This sequence of fetch, decode, execute, and update continues until the program's end, at which point the CPU halts its execution. The stored program concept allows for flexibility, as different programs can be loaded and executed without modifying the computer's hardware. The instructions and data are stored in memory, and the CPU operates on them sequentially based on the program counter.

General System Architecture

In Computer Architecture, the General System Architecture is divided into two major classification units.

1. Store Program Control Concept
2. Flynn's Classification of Computers




Flynn's Classification of Computers
SISD - Single Instruction Single Data
SIMD - Single Instruction Multiple Data
MISD - Multiple Instruction Single Data
MIMD - Multiple Instruction Multiple Data

Von-Neumann Model

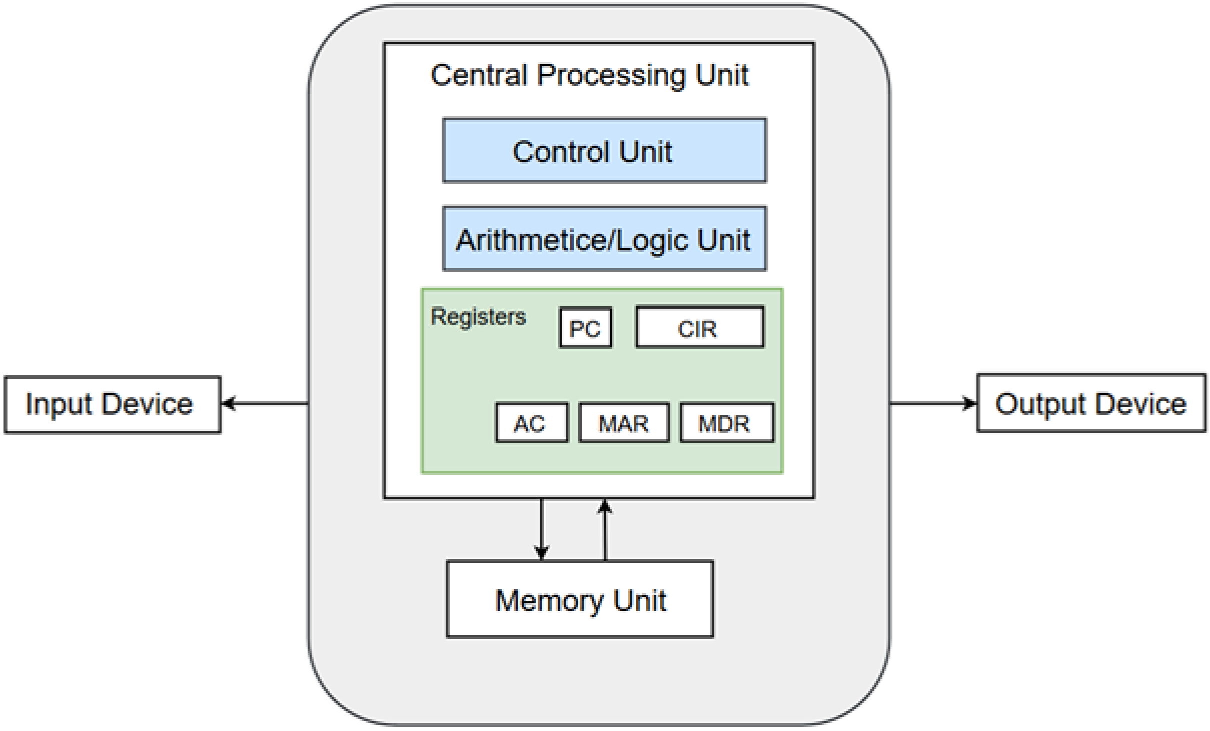
Von-Neumann proposed his computer architecture design in 1945 which was later known as Von-Neumann Architecture. It consisted of a Control Unit, Arithmetic, and Logical Memory Unit (ALU), Registers and Inputs/Outputs.

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.

A Von Neumann-based computer:

- Uses a single processor
- Uses one memory for both instructions and data.
- Executes programs following the fetch-decode-execute cycle

Von-Neumann Basic Structure:



Components of Von-Neumann Model:

- Central Processing Unit
- Buses
- Memory Unit

Central Processing Unit

The part of the Computer that performs the bulk of data processing operations is called the Central Processing Unit and is referred to as the CPU.

The Central Processing Unit can also be defined as an electric circuit responsible for executing the instructions of a computer program.

The CPU performs a variety of functions dictated by the type of instructions that are incorporated in the computer.

The major components of CPU are Arithmetic and Logic Unit (ALU), Control Unit (CU) and a variety of registers.

Arithmetic and Logic Unit (ALU)

The Arithmetic and Logic Unit (ALU) performs the required micro-operations for executing the instructions. In simple words, ALU allows arithmetic (add, subtract, etc.) and logic (AND, OR, NOT, etc.) operations to be carried out.

Control Unit

The Control Unit of a computer system controls the operations of components like ALU, memory and input/output devices.

The Control Unit consists of a program counter that contains the address of the instructions to be fetched and an instruction register into which instructions are fetched from memory for execution.

Registers

Registers refer to high-speed storage areas in the CPU. The data processed by the CPU are fetched from the registers.

Following is the list of registers that plays a crucial role in data processing.

Registers	Description
MAR (Memory Address Register)	This register holds the memory location of the data that needs to be accessed.
MDR (Memory Data Register)	This register holds the data that is being transferred to or from memory.
AC (Accumulator)	This register holds the intermediate arithmetic and logic results.
PC (Program Counter)	This register contains the address of the next instruction to be executed.
CIR (Current Instruction Register)	This register contains the current instruction during processing.

Buses

Buses are the means by which information is shared between the registers in a multiple-register configuration system.

A bus structure consists of a set of common lines, one for each bit of a register, through which binary information is transferred one at a time. Control signals determine which register is selected by the bus during each particular register transfer.

Von-Neumann Architecture comprised of three major bus systems for data transfer.

Bus	Description
Address Bus	Address Bus carries the address of data (but not the data) between the processor and the memory.
Data Bus	Data Bus carries data between the processor, the memory unit and the input/output devices.

Control Bus	Control Bus carries signals/commands from the CPU.

Memory Unit

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.

Two major types of memories are used in computer systems:

1. RAM (Random Access Memory)
2. ROM (Read-Only Memory)

Flynn's Classification of Computers

M.J. Flynn proposed a classification for the organization of a computer system by the number of instructions and data items that are manipulated simultaneously.

The sequence of instructions read from memory constitutes an **instruction stream**.

The operations performed on the data in the processor constitute a **data stream**.

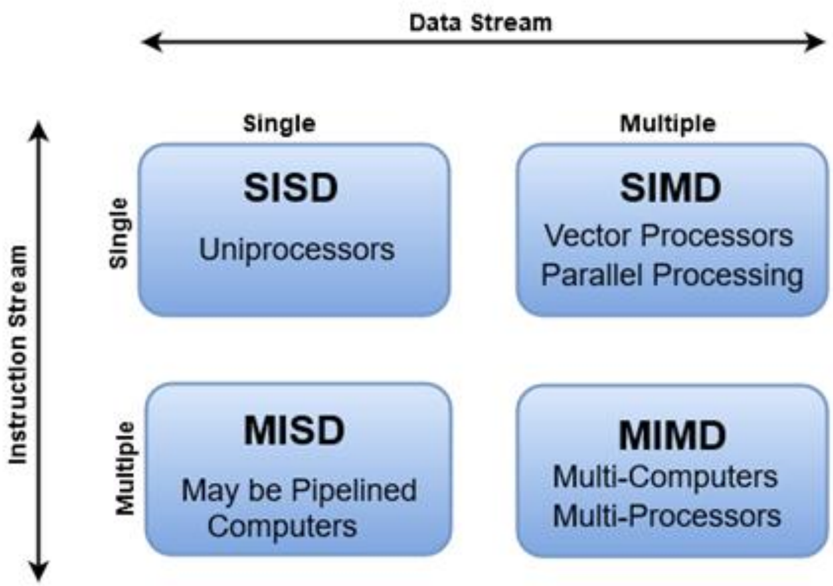
Note: The term 'Stream' refers to the flow of instructions or data.

Parallel processing may occur in the instruction stream, in the data stream, or both.

Flynn's classification divides computers into four major groups that are:

1. Single instruction stream, single data stream (SISD)
2. Single instruction stream, multiple data stream (SIMD)
3. Multiple instruction stream, single data stream (MISD)
4. Multiple instruction stream, multiple data stream (MIMD)

Flynn's Classification of Computers



Types of Computer Architecture

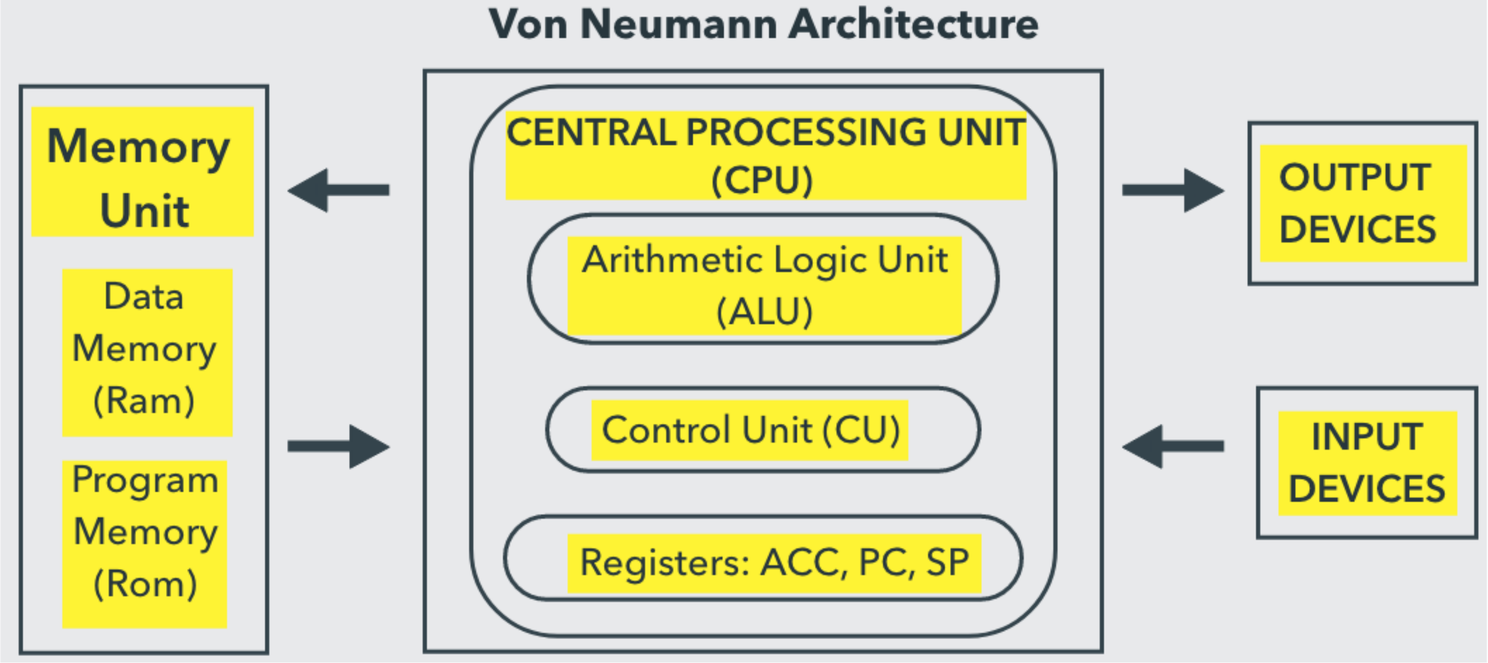
Basically, Microprocessors or Microcontrollers are classified based on the two types of Computer Architecture: Von Neumann Architecture and Harvard Architecture.

Von Neumann Architecture

Von Neumann Architecture or Princeton Architecture is a Computer Architecture, where the Program i.e., the Instructions and the Data are stored in a single memory.

Since the Instruction Memory and the Data Memory are the same, the Processor or CPU cannot access both Instructions and Data at the same time as they use a single bus.

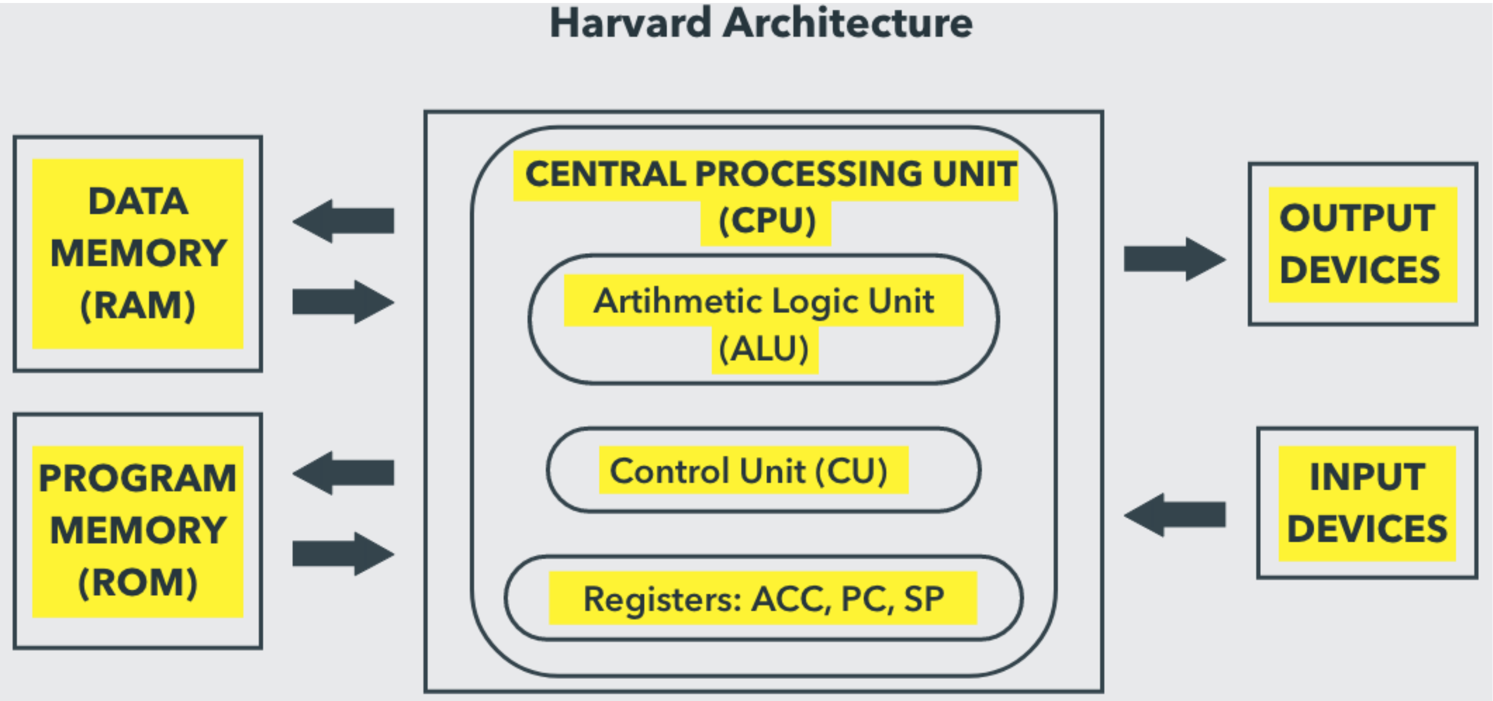
This type of architecture has severe limitations to the performance of the system as it creates a bottleneck while accessing the memory.



Harvard Architecture

Harvard Architecture, in contrast to Von Neumann Architecture, uses separate memory for Instruction (Program) and Data. Since the Instruction Memory and Data Memory are separate in a Harvard Architecture, their signal paths i.e., buses are also different and hence, the CPU can access both Instructions and Data at the same time.

Almost all Microcontrollers, including 8051 Microcontroller implement Harvard Architecture.



Computer Registers

Registers are a type of computer memory used to quickly accept, store, and transfer data and instructions that are being used immediately by the CPU. The registers used by the CPU are often termed as Processor registers.

A processor register may hold an instruction, a storage address, or any data (such as bit sequence or individual characters).

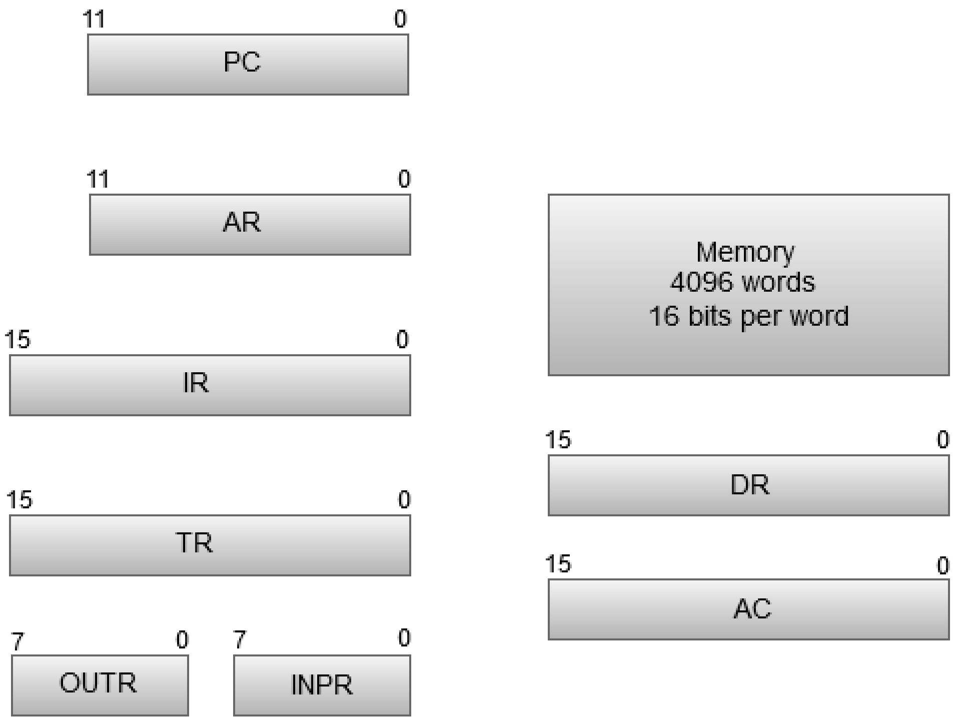
The computer needs processor registers for manipulating data and a register for holding a memory address. The register holding the memory location is used to calculate the address of the next instruction after the execution of the current instruction is completed.

Following is the list of some of the most common registers used in a basic computer:

Register	**Symbol**	**Number of bits**	**Function**
Data register	DR	16	Holds memory operand
Address register	AR	12	Holds address for the memory
Accumulator	AC	16	Processor register
Instruction register	IR	16	Holds instruction code
Program counter	PC	12	Holds address of the instruction
Temporary register	TR	16	Holds temporary data
Input register	INPR	8	Carries input character
Output register	OUTR	8	Carries output character

The following image shows the register and memory configuration for a basic computer.

Register and Memory Configuration of a basic computer:



- The Memory unit has a capacity of 4096 words, and each word contains 16 bits.
- The Data Register (DR) contains 16 bits which hold the operand read from the memory location.
- The Memory Address Register (MAR) contains 12 bits which hold the address for the memory location.
- The Program Counter (PC) also contains 12 bits which hold the address of the next instruction to be read from memory after the current instruction is executed.
- The Accumulator (AC) register is a general purpose processing register.
- The instruction read from memory is placed in the Instruction register (IR).
- The Temporary Register (TR) is used for holding the temporary data during the processing.
- The Input Registers (IR) holds the input characters given by the user.
- The Output Registers (OR) holds the output after processing the input data.

Language Processors: Assembler, Compiler and Interpreter

Computer programs are generally written in high-level languages (like C++, Python, and Java). A language processor, or language translator, is a computer program that convert source code from one programming language to another language or human readable language. They also find errors during translation.

What is Language Processors?

Compilers, interpreters, translate programs written in high-level languages into machine code that a computer understands and assemblers translate programs written in low-level or assembly language into machine code. In the compilation process, there are several stages. To help programmers write error-free code, tools are available.

Assembly language is machine-dependent, yet mnemonics used to represent instructions in it are not directly understandable by machine and high-Level language is machine-independent. A computer understands instructions in machine code, i.e. in the form of 0s and 1s. It is a tedious task to write a computer program directly in machine code. The programs are written mostly in high-level languages like Java, C++, Python etc. and are called source code. These source code cannot be executed directly by the computer and must be converted into machine language to be executed. Hence, a special translator system software is used to translate the program written in a high-level language into machine code is called Language Processor and the program after translated into machine code (object program/object code).

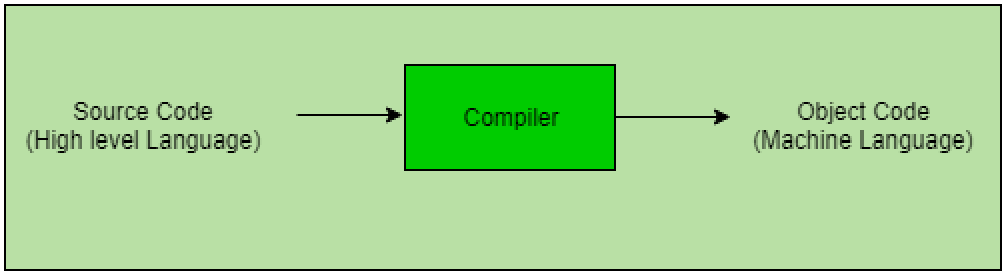
Types of Language Processors

The language processors can be any of the following three types:

1. Compiler

The language processor that reads the complete source program written in high-level language as a whole in one go and translates it into an equivalent program in machine language is called a Compiler. Example: C, C++, C#.

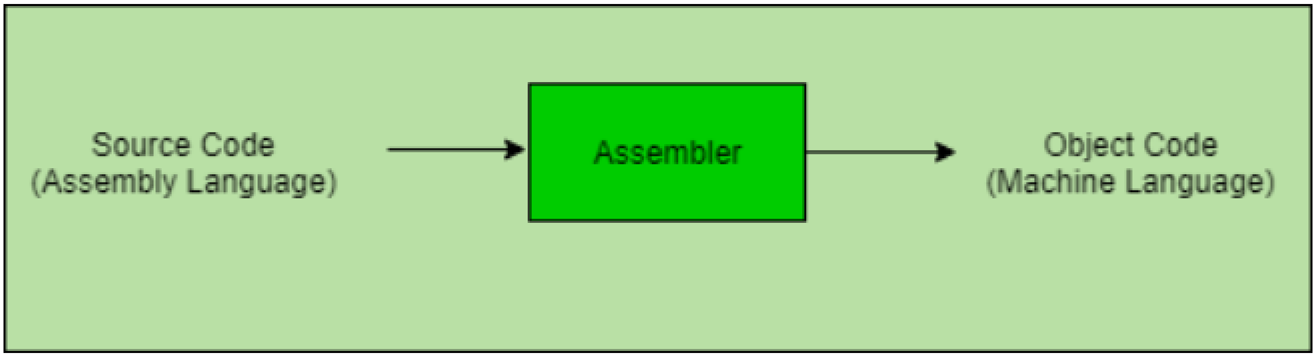
In a compiler, the source code is translated to object code successfully if it is free of errors. The compiler specifies the errors at the end of the compilation with line numbers when there are any errors in the source code. The errors must be removed before the compiler can successfully recompile the source code again the object program can be executed number of times without translating it again.



2. Assembler

The Assembler is used to translate the program written in Assembly language into machine code. The source program is an input of an assembler that contains assembly language instructions. The output generated by the assembler is the object code or machine code understandable by the computer. Assembler is basically the 1st interface that is able to communicate humans with the machine. We need an assembler to fill the gap between human and machine so that they can communicate with each other. code written in assembly language is some sort of mnemonics(instructions) like ADD, MUL, MUX, SUB, DIV, MOV and so on. and the assembler is basically able to convert these mnemonics in binary code. Here, these mnemonics also depend upon the architecture of the machine.

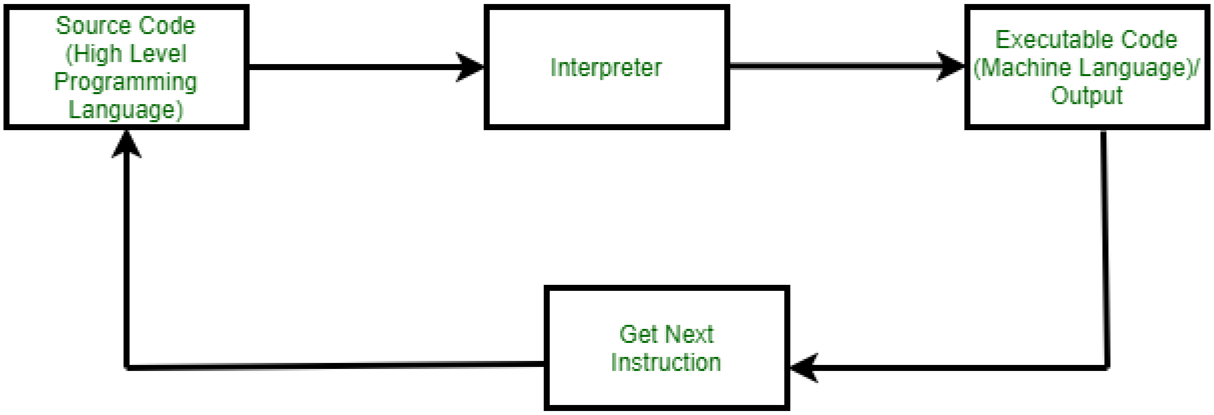
For example, the architecture of intel 8085 and intel 8086 are different.



3. Interpreter

The translation of a single statement of the source program into machine code is done by a language processor and executes immediately before moving on to the next line is called an interpreter. If there is an error in the statement, the interpreter terminates its translating process at that statement and displays an error message. The interpreter moves on to the next line for execution only after the removal of the error. An Interpreter directly executes instructions written in a programming or scripting language without previously converting them to an object code or machine code. An interpreter translates one line at a time and then executes it.

Example: Perl, Python and Matlab.


Source Code (High Level Programming Language)
Interpreter
Executable Code (Machine Language)/ Output
Get Next Instruction

Difference Between Compiler and Interpreter

Compiler	Interpreter
A compiler is a program that converts the entire source code of a programming language into executable machine code for a CPU.	An interpreter takes a source program and runs it line by line, translating each line as it comes to it.
The compiler takes a large amount of time to analyze the entire source code but the overall execution time of the program is comparatively faster.	An interpreter takes less amount of time to analyze the source code but the overall execution time of the program is slower.
The compiler generates the error message only after scanning the whole program, so debugging is comparatively hard as the error can be present anywhere in the program.	Its Debugging is easier as it continues translating the program until the error is met.
The compiler requires a lot of memory for generating object codes.	It requires less memory than a compiler because no object code is generated.
Generates intermediate object code.	No intermediate object code is generated.
For Security purpose compiler is more useful.	The interpreter is a little vulnerable in case of security.

Compiler	Interpreter
Examples: C, C++, C#	Examples: Python, Perl, JavaScript, Ruby.

Role of operating systems and compiler/assembler

Operating systems and compilers/assemblers play crucial roles in the development and execution of software on a computer. Let's explore the functions and roles of each:

Operating System:

1. **Resource Management:**
   - **Memory Management:** Allocates and deallocates memory for programs, ensuring efficient utilization.
   - **Process Management:** Manages processes and threads, facilitating multitasking and concurrency.
   - **File System Management:** Controls access to files and directories, handling storage and retrieval.
2. **Device Management:**
   - Manages communication with hardware devices such as keyboards, printers, and storage devices.
   - Provides a uniform interface for application programs to interact with hardware.
3. **User Interface:**
   - Provides a user interface (UI) through which users interact with the computer.
   - Manages input and output operations, including graphical user interfaces (GUIs) and command-line interfaces (CLIs).
4. **Security:**
   - Implements security mechanisms to protect data, prevent unauthorized access, and ensure the integrity of the system.
5. **Networking:**
   - Facilitates communication between computers over a network.
   - Manages network connections, protocols, and data transmission.
6. **Error Handling:**
   - Detects and handles errors that may occur during the execution of programs.
   - Provides a mechanism for reporting errors to users or logging them for system administrators.
7. **Job Scheduling:**
   - Allocates CPU time to different processes and manages the execution of multiple tasks concurrently.
8. **File System:**
   - Provides a hierarchical structure for organizing and storing files on storage devices.
   - Implements file access controls, permissions, and file attributes.

Compiler/Assembler:

1. **Compilation/Assembly:**
   - **Compiler:** Translates high-level programming code (e.g., C, C++) into machine code or an intermediate code.
   - **Assembler:** Translates assembly language code into machine code.
2. **Optimization:**
   - Analyzes and optimizes code for better performance, making efficient use of system resources.
   - Generates optimized machine code to improve the speed and efficiency of the compiled program.

3. **Error Checking:**
    - Identifies syntax and semantic errors in the source code during the compilation process.
    - Provides error messages to help programmers correct mistakes before execution.
4. **Code Generation:**
    - Transforms the source code into executable machine code that can be understood and executed by the computer's CPU.
5. **Linking:**
    - Combines separately compiled modules and resolves references between them to create a complete executable program.
    - May include linking external libraries and modules.
6. **Debugging Support:**
    - Generates debugging information to assist developers in identifying and fixing issues in their code.
    - Provides tools for source-level debugging.
7. **Portability:**
    - Allows programs to be written in high-level languages, making them portable across different platforms without modification.

In summary, the operating system provides a foundation for the execution of programs, managing resources and interactions with hardware, while compilers and assemblers translate high-level source code into machine code and assist in the optimization and linking of programs. Together, these components contribute to the development and execution of software on a computer system.

Fetch, decode, and execute cycle

The fetch, decode, and execute cycle is a fundamental concept in the operation of a computer's central processing unit (CPU). This cycle is repeated continuously during the execution of a computer program. Here's an overview of each step in the cycle:

1. Fetch:

- **Fetch Instruction:**
    - The CPU fetches the next instruction from the memory. The address of the instruction to be fetched is determined by the program counter (PC).
    - The program counter is a special register that keeps track of the memory address of the next instruction to be executed.
- **Memory Access:**
    - The CPU sends a request to the memory subsystem to read the instruction stored at the memory address indicated by the program counter.
- **Update Program Counter:**
    - After fetching the instruction, the program counter is incremented to point to the next instruction in sequence.

2. Decode:

- **Instruction Decoding:**
    - The fetched instruction is decoded by the CPU's control unit.
    - The control unit determines the operation to be performed, the operands involved, and any necessary addressing modes.

3. Execute:

- **Instruction Execution:**
    - The CPU performs the operation specified by the decoded instruction.

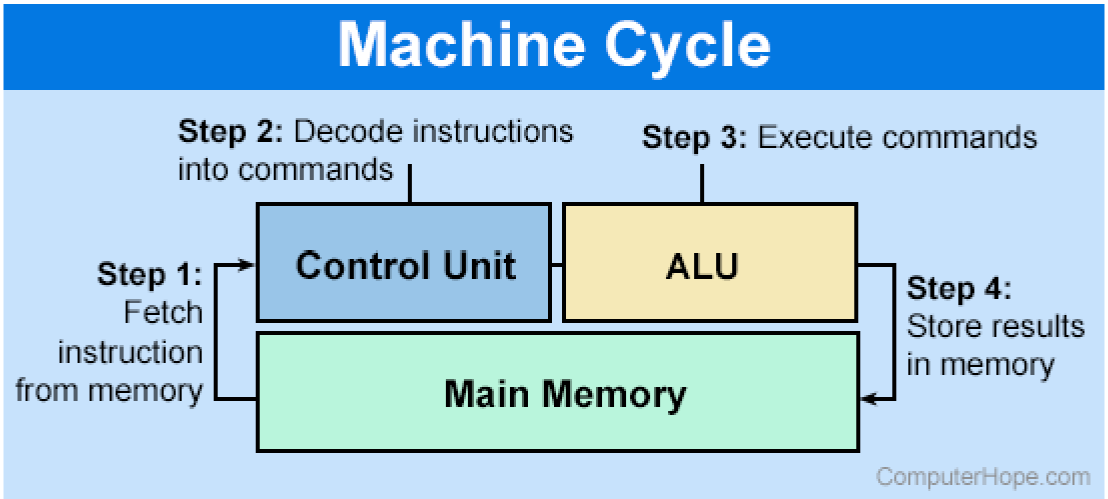
- This may involve arithmetic or logic operations, data manipulation, or control flow changes.

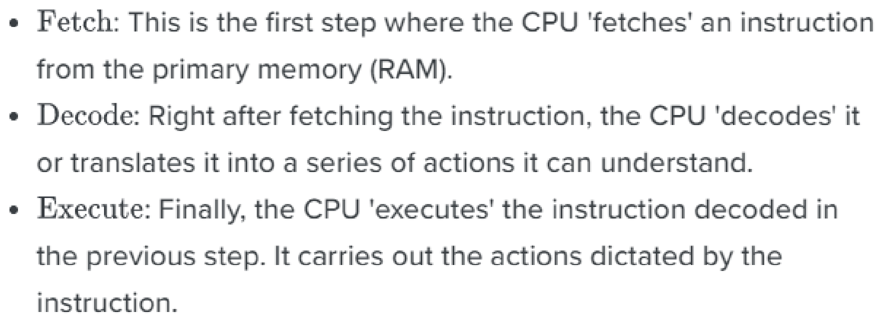
- **Data Movement:**
  - If the instruction involves moving data, the CPU may read from or write to registers, memory locations, or I/O devices.

4. Repeat:

- **Update Program Counter:**
  - The program counter is updated to point to the next instruction in memory, preparing for the next cycle.
  - The cycle repeats, starting with the fetch phase for the next instruction.

The fetch, decode, and execute cycle continues until the program is complete. Each iteration of the cycle processes one instruction at a time, and the sequence of instructions is determined by the program being executed.


Machine Cycle
Step 2: Decode instructions into commands
Step 3: Execute commands
Step 1: Fetch instruction from memory
Control Unit
ALU
Step 4: Store results in memory
Main Memory
ComputerHope.com



- Fetch: This is the first step where the CPU 'fetches' an instruction from the primary memory (RAM).
- Decode: Right after fetching the instruction, the CPU 'decodes' it or translates it into a series of actions it can understand.
- Execute: Finally, the CPU 'executes' the instruction decoded in the previous step. It carries out the actions dictated by the instruction.

---

instruction and instruction format

An instruction in computer science refers to a binary code that represents a specific operation to be performed by the central processing unit (CPU) of a computer. Instructions are the basic building blocks of a computer program, and they dictate the tasks the CPU must carry out. Each instruction corresponds to a specific operation, such as arithmetic, logic, data movement, or control flow.

Instructions typically consist of two main components:

1. **Opcode (Operation Code):**

- The opcode is a binary code that specifies the operation or task to be performed by the CPU. It indicates the type of instruction, such as addition, subtraction, loading data from memory, or branching to a different part of the program.

2. **Operands:**
    - Operands are the data or addresses on which the operation specified by the opcode is to be performed. The number and type of operands can vary depending on the instruction and the computer architecture.

Instruction Format:

The instruction format refers to the organization of bits within a machine language instruction. It defines how the opcode and operands are represented in the binary code. Different computer architectures may have different instruction formats. Common formats include:

1. **Fixed-Length Format:**
    - In this format, each instruction has a fixed number of bits for the opcode and operands. The length is determined by the architecture of the computer.
    - Example: The MIPS (Microprocessor without Interlocked Pipeline Stages) architecture uses a fixed-length instruction format.
2. **Variable-Length Format:**
    - In this format, the length of the instruction can vary. The opcode may be followed by a variable number of bits representing operands.
    - Example: The x86 architecture often uses a variable-length instruction format.
3. **Operand Address Format:**
    - The instruction specifies the address of the operands in memory. This format is common in architectures that use explicit memory addresses in their instructions.
    - Example: The von Neumann architecture often employs operand address formats.
4. **Register-Register Format:**
    - Both the opcode and the operands specify register numbers. This format is common in architectures that heavily use registers for operations.
    - Example: The RISC (Reduced Instruction Set Computing) architecture often uses a register-register format.

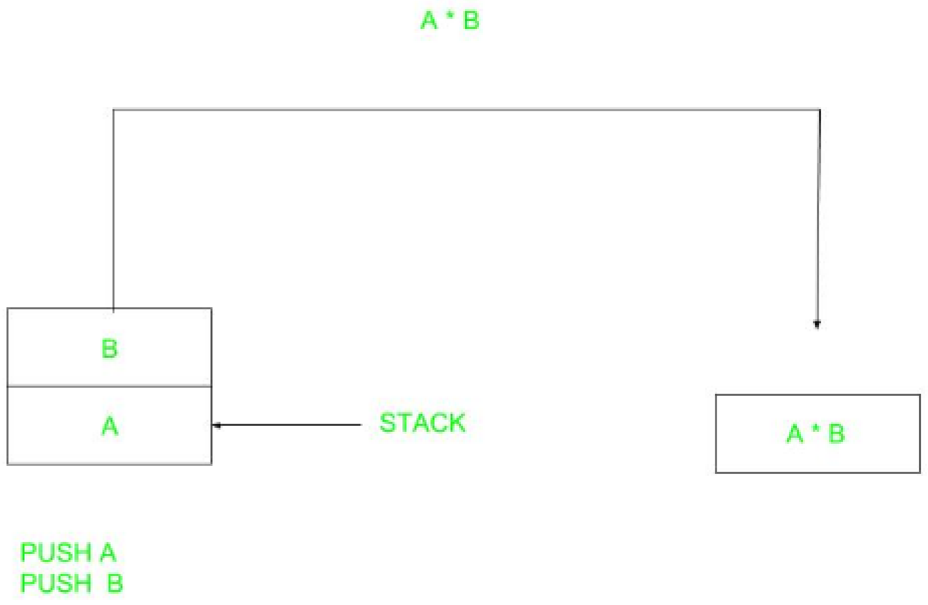
Types of Instructions

Based on the number of addresses, instructions are classified as:

`NOTE:  We will use the X = (A+B)*(C+D) expression to showcase the procedure.`

Zero Address Instructions

These instructions do not specify any operands or addresses. Instead, they operate on data stored in registers or memory locations implicitly defined by the instruction. For example, a zero-address instruction might simply add the contents of two registers together without specifying the register names.



Zero Address Instruction

Zero Address Instruction

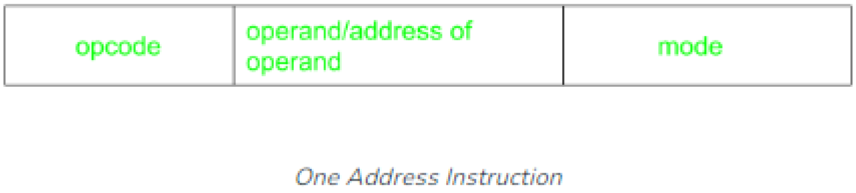
A stack-based computer does not use the address field in the instruction. To evaluate an expression first it is converted to reverse Polish Notation i.e. Postfix Notation.

```
Expression: X = (A+B)*(C+D)
Postfixed : X = AB+CD+*
TOP means top of stack
M[X] is any memory location
```

PUSH	A	TOP = A
PUSH	B	TOP = B
ADD		TOP = A+B
PUSH	C	TOP = C
PUSH	D	TOP = D
ADD		TOP = C+D
MUL		TOP = (C+D)*(A+B)
POP	X	M[X] = TOP

One Address Instructions

These instructions specify one operand or address, which typically refers to a memory location or register. The instruction operates on the contents of that operand, and the result may be stored in the same or a different location. For example, a one-address instruction might load the contents of a memory location into a register.
This uses an implied ACCUMULATOR register for data manipulation. One operand is in the accumulator and the other is in the register or memory location. Implied means that the CPU already knows that one operand is in the accumulator so there is no need to specify it.



opcode	operand/address of operand	mode

One Address Instruction

One Address Instruction

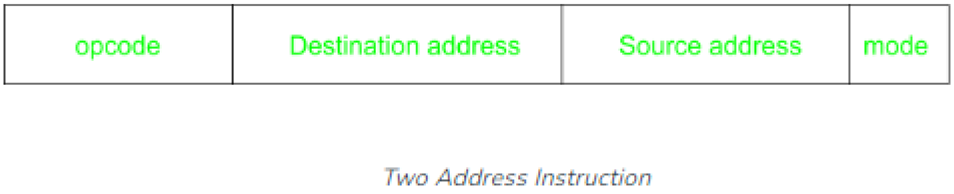
```
Expression: X = (A+B)*(C+D)
AC is accumulator
M[] is any memory location
M[T] is temporary location
```

LOAD	A	AC = M[A]
ADD	B	AC = AC + M[B]
STORE	T	M[T] = AC
LOAD	C	AC = M[C]
ADD	D	AC = AC + M[D]
MUL	T	AC = AC * M[T]
STORE	X	M[X] = AC

Two Address Instructions

These instructions specify two operands or addresses, which may be memory locations or registers. The instruction operates on the contents of both operands, and the result may be stored in the same or a different location. For example, a two-address instruction might add the contents of two registers together and store the result in one of the registers.
This is common in commercial computers. Here two addresses can be specified in the instruction. Unlike earlier in one address instruction, the result was stored in the accumulator, here the result

can be stored at different locations rather than just accumulators, but require more number of bit to represent the address.



opcode	Destination address	Source address	mode

Two Address Instruction

Here destination address can also contain an operand.

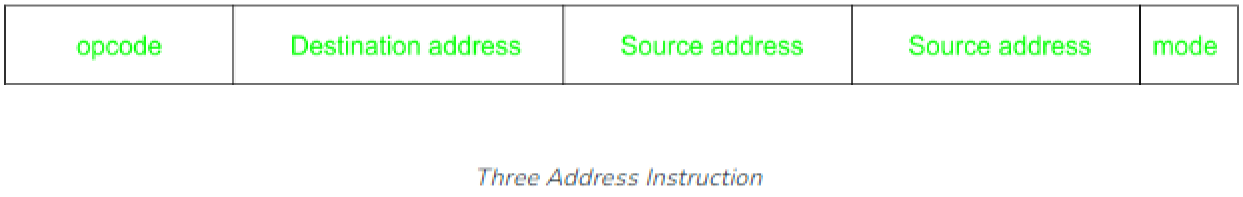
```
Expression: X = (A+B)*(C+D)
R1, R2 are registers
M[] is any memory location
```

MOV	R1, A	R1 = M[A]
ADD	R1, B	R1 = R1 + M[B]
MOV	R2, C	R2 = M[C]
ADD	R2, D	R2 = R2 + M[D]
MUL	R1, R2	R1 = R1 * R2
MOV	X, R1	M[X] = R1

Three Address Instructions

These instructions specify three operands or addresses, which may be memory locations or registers. The instruction operates on the contents of all three operands, and the result may be stored in the same or a different location. For example, a three-address instruction might multiply the contents of two registers together and add the contents of a third register, storing the result in a fourth register.

This has three address fields to specify a register or a memory location. Programs created are much short in size but number of bits per instruction increases. These instructions make the creation of the program much easier but it does not mean that program will run much faster because now instructions only contain more information but each micro-operation (changing the content of the register, loading address in the address bus etc.) will be performed in one cycle only.



opcode	Destination address	Source address	Source address	mode

Three Address Instruction

Three Address Instruction

```
Expression: X = (A+B)*(C+D)
R1, R2 are registers
M[] is any memory location
```

ADD	R1, A, B	R1 = M[A] + M[B]
ADD	R2, C, D	R2 = M[C] + M[D]
MUL	X, R1, R2	M[X] = R1 * R2

Advantages of Zero-Address, One-Address, Two-Address and Three-Address Instructions

Zero-address instructions

- They are simple and can be executed quickly since they do not require any operand fetching or addressing. They also take up less memory space.

One-address instructions

- They allow for a wide range of addressing modes, making them more flexible than zero-address instructions. They also require less memory space than two or three-address instructions.

Two-address instructions

- They allow for more complex operations and can be more efficient than one-address instructions since they allow for two operands to be processed in a single instruction. They also allow for a wide range of addressing modes.

Three-address instructions

- They allow for even more complex operations and can be more efficient than two-address instructions since they allow for three operands to be processed in a single instruction. They also allow for a wide range of addressing modes.

Disadvantages of Zero-Address, One-Address, Two-Address and Three-Address Instructions

Zero-address instructions

- They can be limited in their functionality and do not allow for much flexibility in terms of addressing modes or operand types.

One-address instructions

- They can be slower to execute since they require operand fetching and addressing.

Two-address instructions

- They require more memory space than one-address instructions and can be slower to execute since they require operand fetching and addressing.

Three-address instructions

- They require even more memory space than two-address instructions and can be slower to execute since they require operand fetching and addressing.

Addressing Modes-

> The different ways of specifying the location of an operand in an instruction are called as **addressing modes**.

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 specify the operands implicitly.
- It is also called as **implicit addressing mode**.

Examples-

- The instruction “Complement Accumulator” is an implied mode instruction.
- In a stack organized computer, Zero Address Instructions are implied mode instructions.

(since operands are always implied to be present on the top of the stack)

2. Stack Addressing Mode-

In this addressing mode,

- The operand is contained at the top of the stack.

Example-

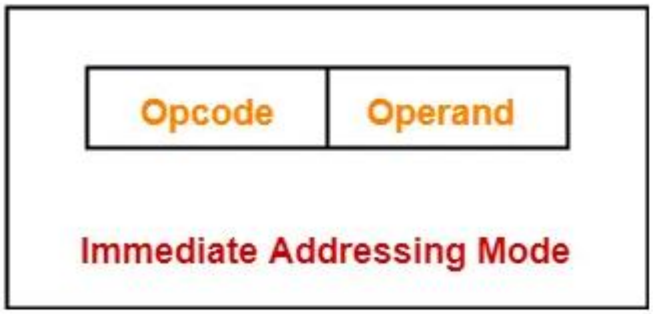
ADD

- 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.



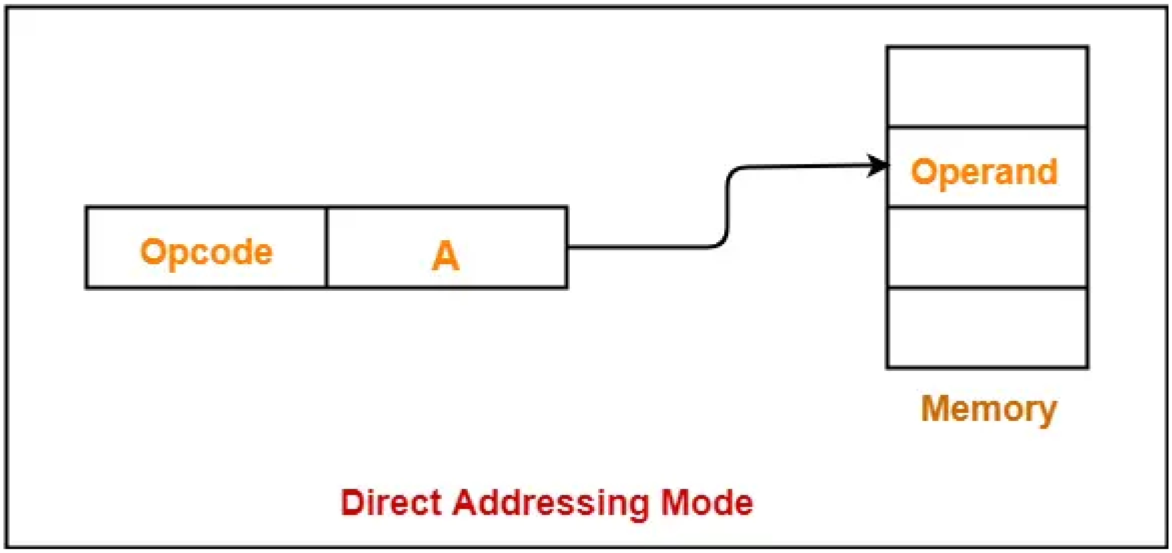
Examples-

- ADD 10 will increment the value stored in the accumulator by 10.
- MOV R #20 initializes 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-

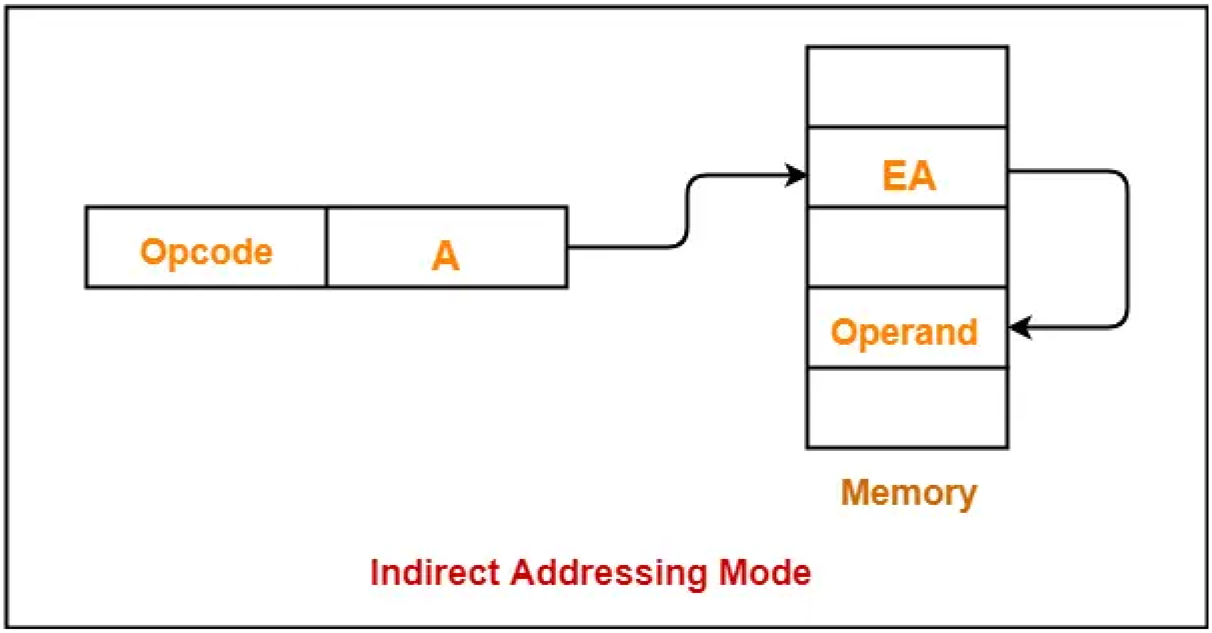
- ADD X will increment the value stored in the accumulator by the value stored at memory location X.

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.



Example-

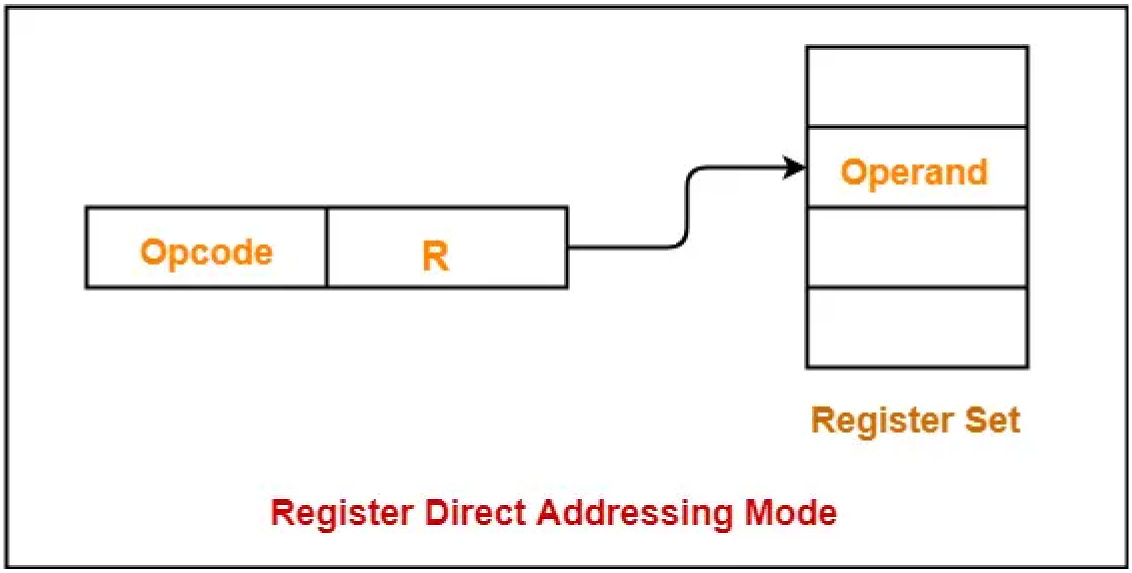
- ADD X will increment the value stored in the accumulator by the value stored at memory location specified by X.

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]

NOTE-

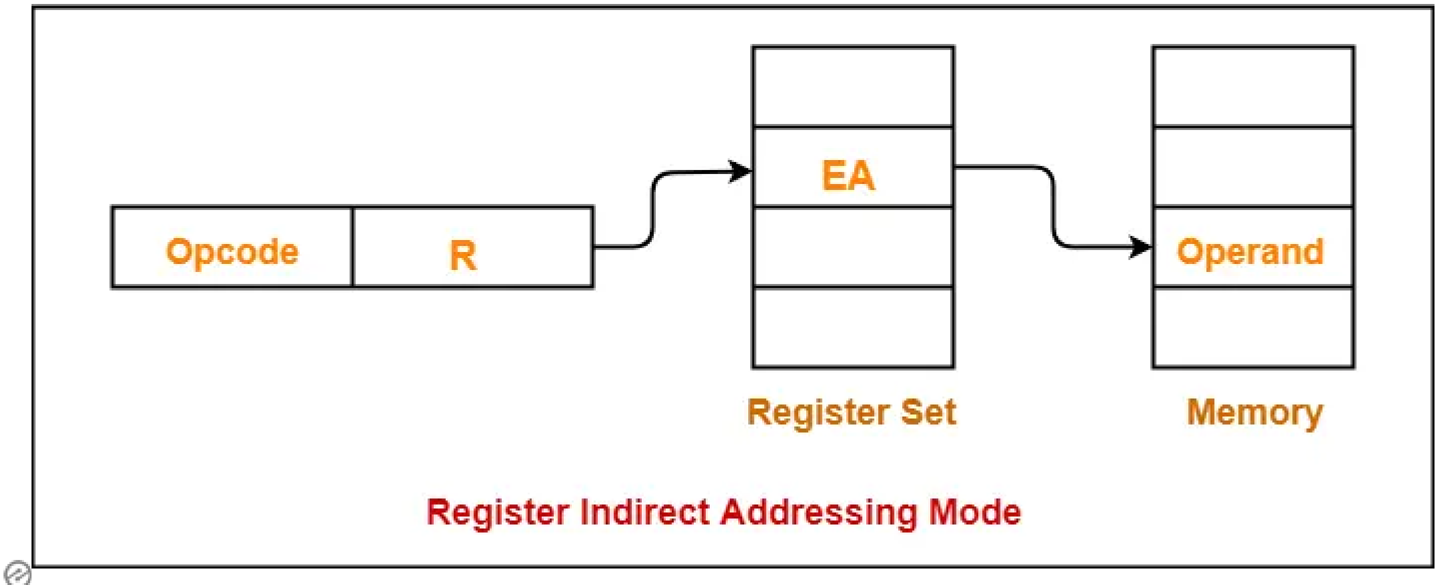
It is interesting to note-

- 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.

$$AC \leftarrow AC + [[R]]$$

NOTE-

It is interesting to note-

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

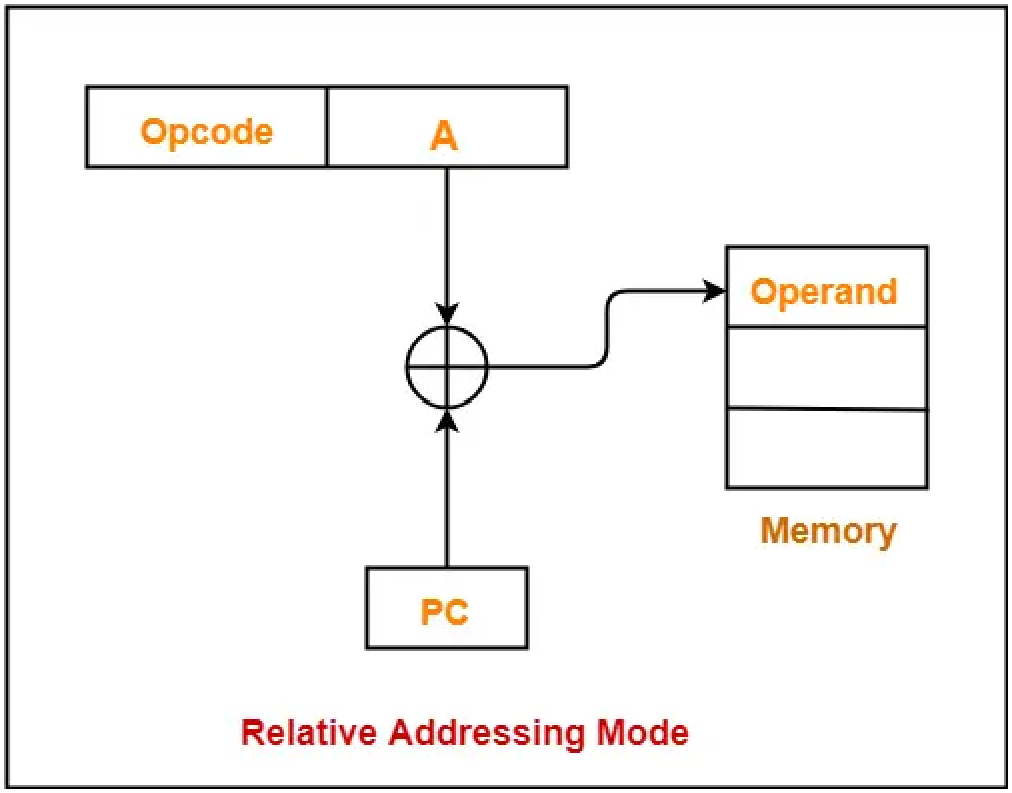
8. Relative Addressing Mode-

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



NOTE-

- **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.

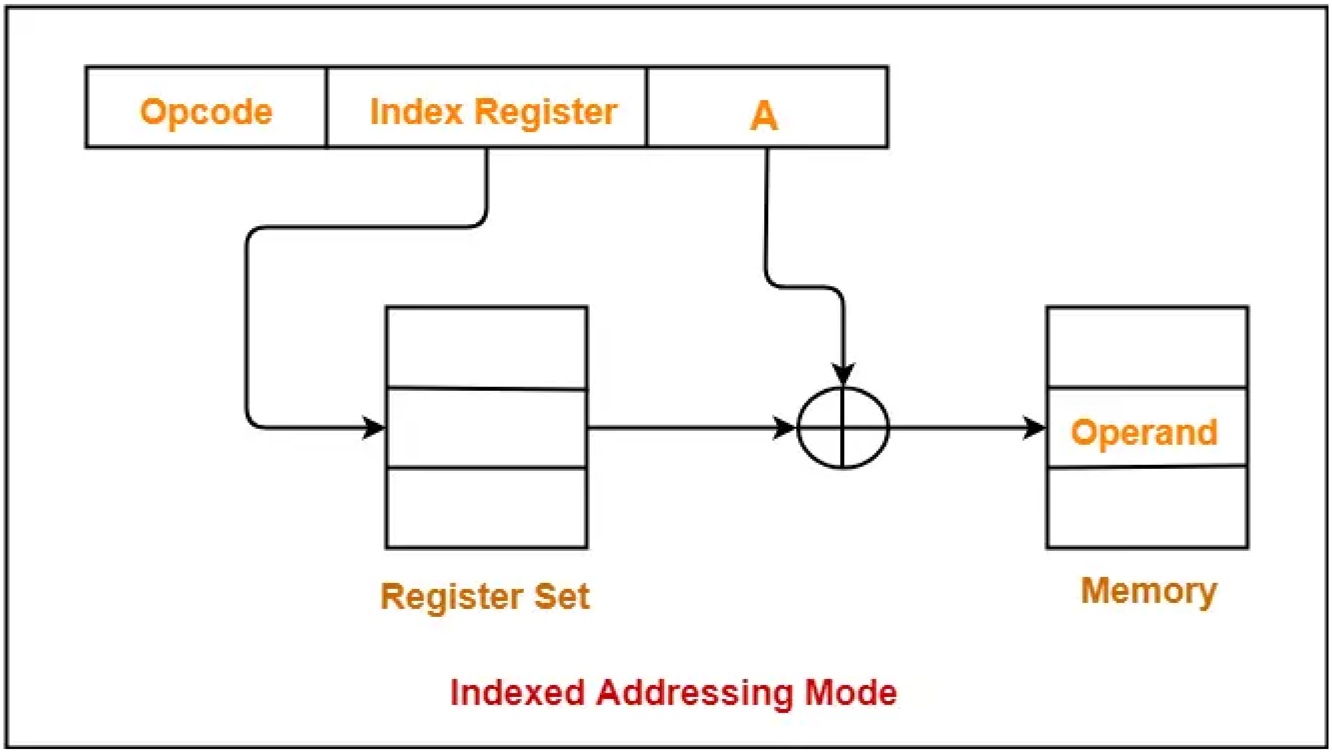
9. Indexed Addressing Mode-

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



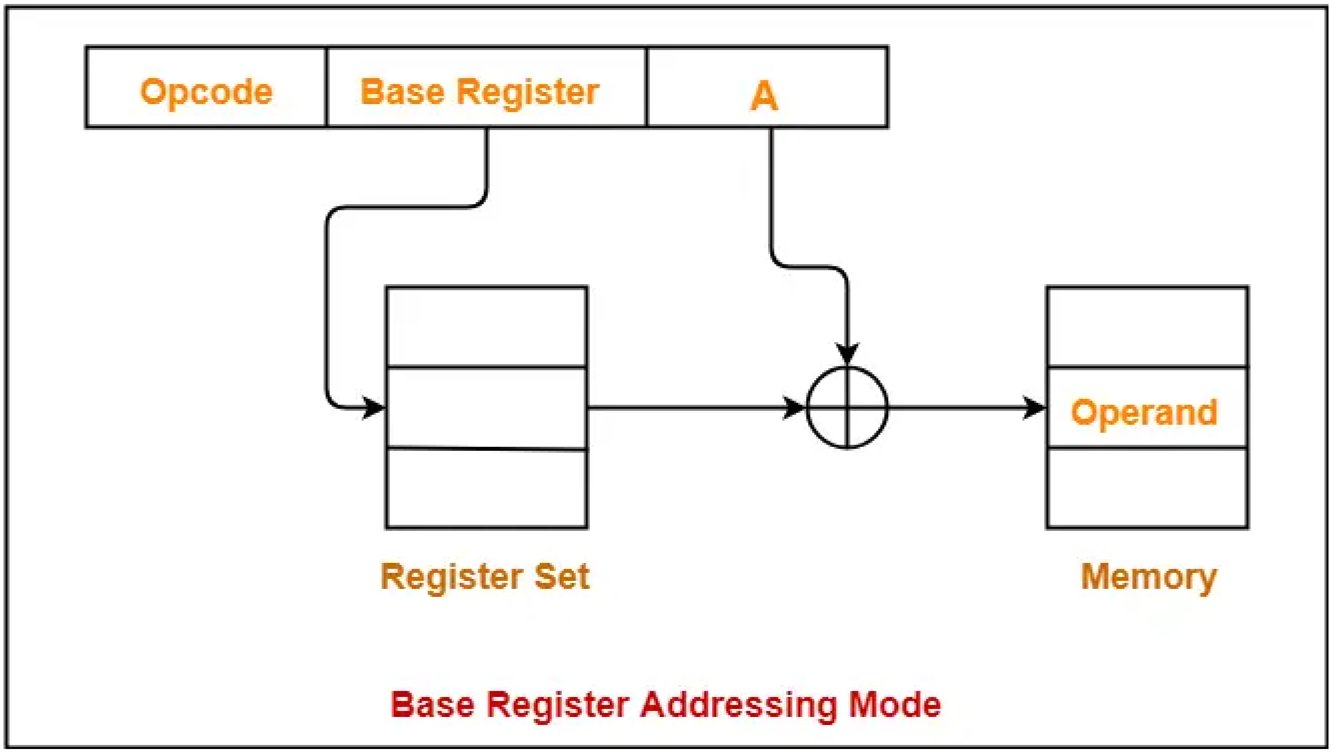
10. Base Register Addressing Mode-

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-

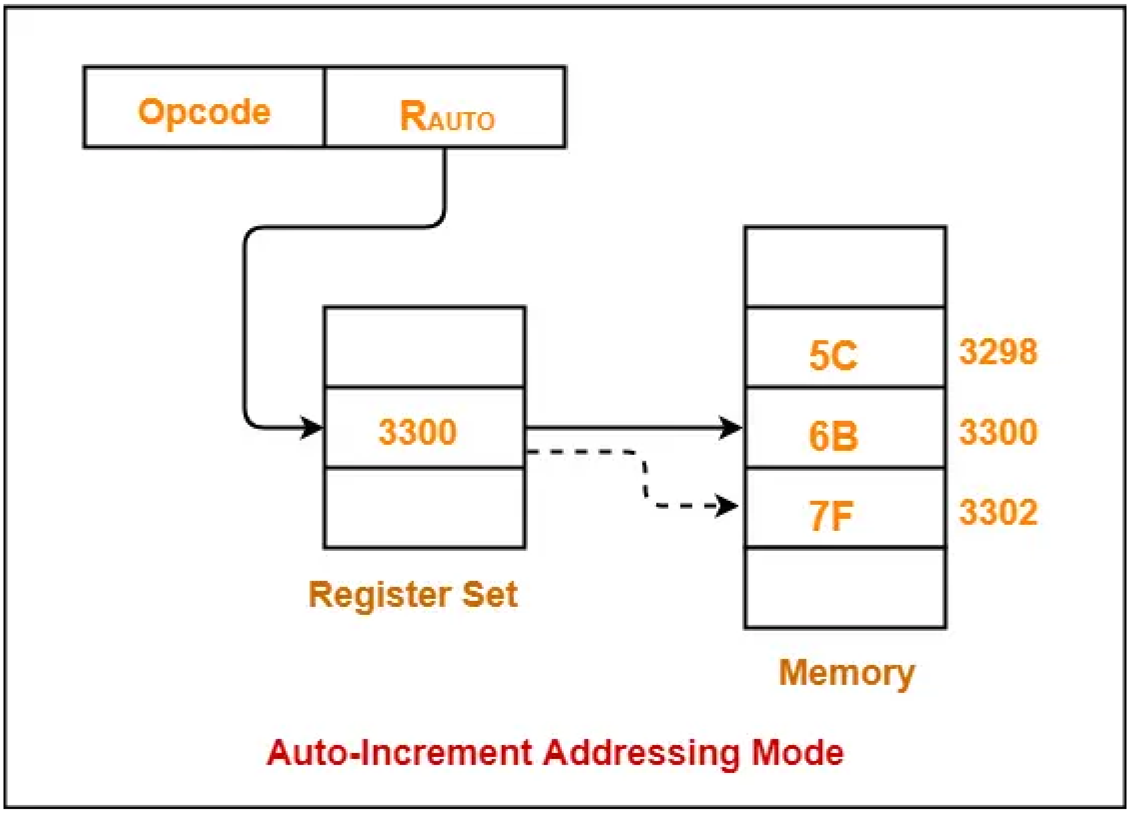
Effective Address of the Operand

= 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-



Auto-Increment Addressing Mode

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-

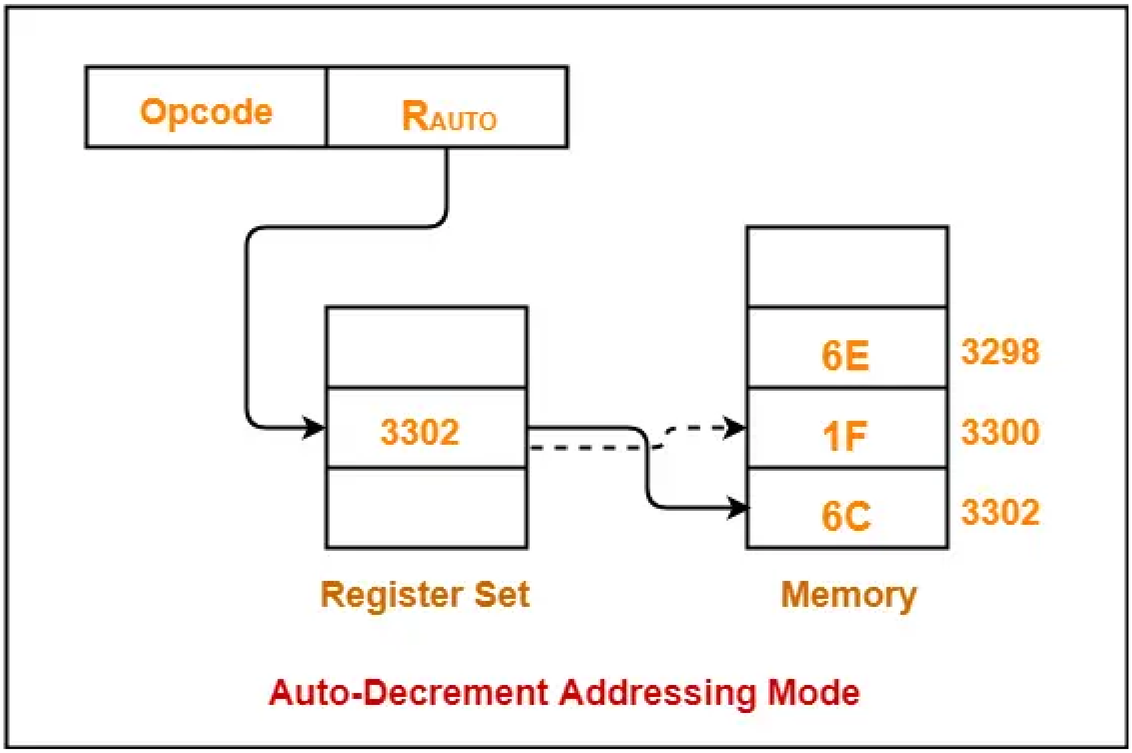
Effective Address of the Operand

= 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.

Fixed and floating point representation of numbers

Fixed-point and floating-point representations are methods used in computing to represent numbers with different characteristics. Let's explore each of them:

Fixed-Point Representation:

Definition:

- In fixed-point representation, a specific number of bits are allocated to the integer and fractional parts of a number.
- The position of the binary point (or decimal point in the case of decimal fixed-point) is fixed, and numbers are represented with a constant number of fractional and integer bits.

Characteristics:

1. **Precision:**
    - Fixed-point representation provides fixed precision. The number of fractional bits determines the precision, but it is constant for all numbers.
2. **Range:**
    - The range of representable values is fixed by the total number of bits available. Overflow or underflow may occur if the result exceeds the representable range.
3. **Complexity:**
    - Fixed-point arithmetic is generally simpler and computationally less expensive than floating-point arithmetic.
4. **Applications:**
    - Commonly used in applications where a fixed range of values with a specific precision is sufficient, such as digital signal processing.

Floating-Point Representation:

Definition:

- In floating-point representation, a number is represented as a sign, a significand (also called mantissa), and an exponent. The position of the binary point (or decimal point) is not fixed but is determined by the exponent.

Characteristics:

1. **Precision:**
    - Floating-point representation provides variable precision. The precision can be adjusted based on the exponent, allowing a wide range of magnitudes with consistent relative precision.
2. **Range:**
    - The range of representable values is determined by the exponent. Floating-point representation can handle a broader range of values compared to fixed-point.
3. **Complexity:**
    - Floating-point arithmetic is more complex than fixed-point arithmetic, and it typically requires specialized hardware (e.g., floating-point unit) for efficient computation.
4. **Applications:**
    - Commonly used in applications where a wide range of values and variable precision are required, such as scientific calculations and graphics processing.

Example:

Consider the decimal number 123.456.

- **Fixed-Point Representation (assuming 8 bits for integer and 8 bits for fractional part):**
    - 01111011.01110100
- **Floating-Point Representation (single-precision IEEE 754 format):**
    - Sign: 0 (positive)
    - Exponent: 10000010 (biased exponent, equivalent to 130 in decimal)
    - Significand: 1111011.01110100000000000000 (normalized binary representation)

In summary, fixed-point representation has a fixed number of fractional and integer bits, providing constant precision, while floating-point representation allows for variable precision and a broader range of values through the use of an exponent. The choice between fixed-point and floating-point depends on the requirements of the specific application.