UNIT – 4 : CENTRAL PROCESSING UNIT

Introduction:

The main 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**.

Its purpose is to interpret instruction cycles received from memory and perform arithmetic, logic and control operations with data stored in internal register, memory words and I/O interface units.

The CPU is made up of three major parts, as shown in Fig. –

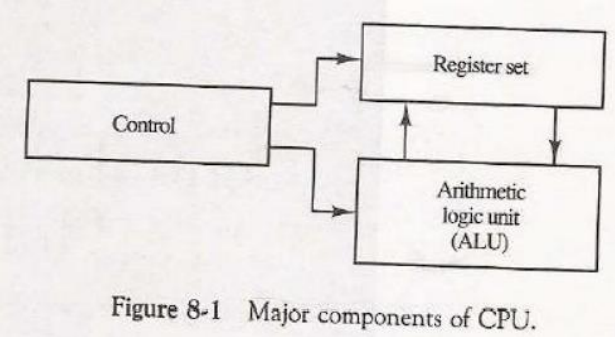


Figure 8-1 Major components of CPU.

- The **register set** stores intermediate data used during the execution of the instructions.

- The **arithmetic logic unit (ALU)** performs the required microoperations for executing the instructions.

- The **control unit** supervises the transfer of information among the registers and instructs the ALU as to which operation to perform.

The operation or task that must perform by CPU is:

• **Fetch Instruction:** The CPU reads an instruction from memory.

• **Interpret Instruction:** The instruction is decoded to determine what action is required.

• **Fetch Data:** The execution of an instruction may require reading data from memory or I/O module.

• **Process data:** The execution of an instruction may require performing some arithmetic or logical operation on data.

• **Write data:** The result of an execution may require writing data to memory or an I/O module.

To do these tasks, it should be clear that the CPU needs to store some data temporarily. It must remember the location of the last instruction so that it can know where to get the next instruction. It needs to store instructions and data temporarily while an

instruction is being executed. In other words, the CPU needs a small internal memory. These storage locations are generally referred as **Registers**.

The CPU is connected to the rest of the system through system bus. Through system bus, data or information gets transferred between the CPU and the other component of the system. The system bus may have three components:

Data Bus: Data bus is used to transfer the data between main memory and CPU.

Address Bus: Address bus is used to access a particular memory location by putting the address of the memory location.

Control Bus: Control bus is used to provide the different control signal generated by CPU to different part of the system.

As for example, memory read is a signal generated by CPU to indicate that a memory read operation has to be performed. Through control bus this signal is transferred to memory module to indicate the required operation.

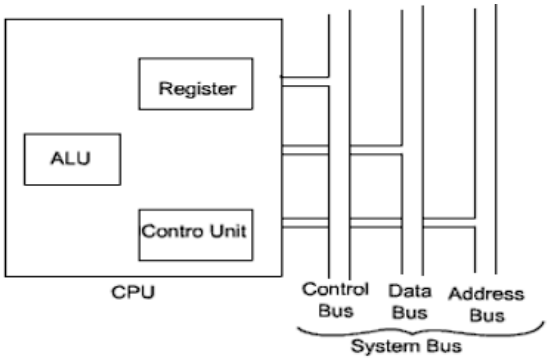


Figure 1: CPU with the system bus.

There are three basic components of CPU: register bank, ALU and Control Unit. There are several data movements between these units and for that an internal CPU bus is used. Internal CPU bus is needed to transfer data between the various registers and the ALU.

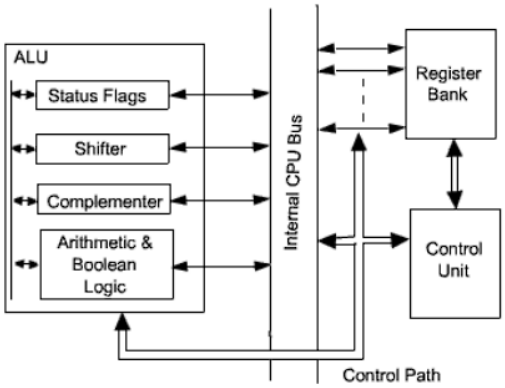


Figure 2 : Internal Structure of CPU

Most computers fall into one of three types of CPU organizations:

1. Single accumulator organization 2. General register organization 3. Stack organization

General Register Organization:

Introduction:

A register is made up of flip-flops. In the CPU (Central Processing Unit), a register is a one-of-a-kind, high-speed storage region. Combinational circuits are used to implement data processing. Before processing, the data is always defined in a register. Program implementation is faster thanks to the registers.

The following are two essential functions implemented by registers in CPU operation:

1. It can be used as a temporary data store site. This allows directly implementing applications to have quick access to data when needed.

2. It can record the CPU's condition and information about the currently executing programme.

What is General Register Organization?

General Register Organization refers to the structure and usage of general-purpose registers within a CPU. These registers are used for various computational and data manipulation tasks during program execution. The organization typically defines the number of registers available, their size, and their specific roles, including temporary data storage, addressing, and operand manipulation. The organization of general registers can vary between different CPU architectures and designs, impacting the CPU's performance and capabilities.

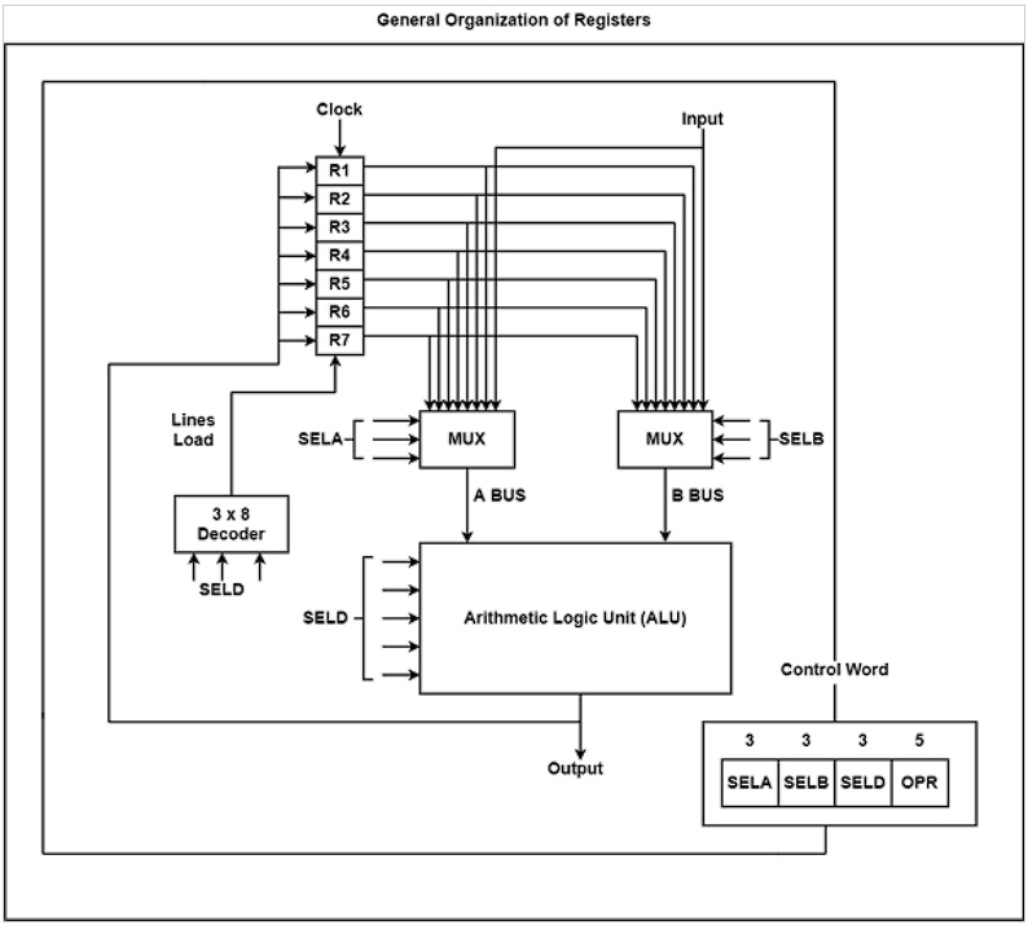
When we are using multiple general-purpose registers, instead of a single accumulator register, in the CPU Organization then this type of organization is known as General register-based CPU Organization. In this type of organization, the computer uses two or three address fields in their instruction format. Each address field may specify a general register or a memory word. If many CPU registers are available for heavily used variables and intermediate results, we can avoid memory references much of the time, thus vastly increasing program execution speed, and reducing program size.

Some examples of General register-based CPU Organizations are **IBM 360 and PDP-11**.

Example:

The registers save the address of the next programme instruction, signals from external devices, error messages, and various data.

If a CPU has some registers, these registers can be linked by a shared bus. The below image depicts the general organization of seven CPU registers -



The control unit is in charge of the CPU bus system. The control unit specifies the data flow via the ALU by selecting the ALU's function and system components.

Let us consider R1←R2 + R3, and the functions implemented within the CPU are as follows:

Function name	**Description**
MUX A Selector (SELA)	It can insert R2 into bus A.
MUX B Selector (SELB)	It can insert R3 in bus B.
ALU Operation Selector (OPR)	It can select the arithmetic addition (ADD).
Decoder Destination Selector (SELD)	It can transfer the result into R1.

The buses are used to perform the multiplexers of 3-state gates. The control word is determined by the status of 14 binary selection inputs. The 14-bit control word defines the micro-operation.

Register Selection Field Encoding:

The table specifies the encoding of register selection fields -

Binary Code	SELA	SELB	SELD
000	Input	Input	None
001	R1	R1	R1
010	R2	R2	R2
011	R3	R3	R3
100	R4	R4	R4
101	R5	R5	R5
110	R6	R6	R6
111	R7	R7	R7

The ALU is in charge of several micro-operations.

ALU Operations Encoding:

The table shows a few of the operations that the ALU performs –

OPR Select	Operation	Symbol
00000	Transfer A	TSFA
00001	Increment A	INCA
00010	Add A + B	ADD
00101	Subtract A - B	SUB
00110	Decrement A	DECA
01000	ADD A and B	AND
01010	OR A and B	OR
01100	XOR A and B	XOR
01110	Complement A	COMA
10000	Shift right A	SHRA
11000	Shift left A	SHLA

ALU Micro-Operations:

In the table, certain ALU micro-operations are listed -

Micro-operation	SELA	SELB	SELD	OPR	Control Word
R1 ← R2 – R3	R2	R3	R1	SUB	010 011 001 00101
R4 ← R4 ∨ R5	R4	R5	R4	OR	100 101 100 01010
R6 ← R6 + R1	-	R6	R1	INCA	110 000 110 00001
R7 ← R1	R1	-	R7	TSFA	001 000 111 00000
Output ← R2	R2	-	None	TSFA	010 000 000 00000
Output ← Input	Input	-	None	TSFA	000 000 000 00000
R4 ← shl R4	R4	-	R4	SHLA	100 000 100 11000
R5 ← 0	R5	R5	R5	XOR	101 101 101 01100

Control Word:

The control word is determined by the sum of the binary selection inputs. It is divided into four sections. SELA, SELB, and SELD each have three bits, and the OPR field has four bits, for a total of 13 bits in the control word.

Format of Control word -

1. The SELA's three bits choose a source register for the ALU's input.

2. The three bits of SELB are used to pick a source register for the ALU's b input.

3. Using the decoder, the three bits of SELD pick a target register.

4. The four bits of OPR determine which operation the ALU will do.

Control word for operation R2 ← R1 + R3.

SELA	SELB	SELD	OPR
001	011	010	0010

Features of a General Register Organization:

A General Register Organization in computer architecture typically includes the following features:

- **Register Set:** It consists of a fixed number of general-purpose registers (GPRs) used for various computational tasks.

- **Register Size:** Registers can have different sizes, typically ranging from 8 to 64 bits, and are used to store data and addresses.
- **Data Storage:** Registers are used for temporary data storage, which helps in performing arithmetic and logic operations efficiently.
- **Operand Manipulation:** They hold operands for mathematical operations, comparisons, and logical operations.
- **Addressing:** Some registers are used to hold memory addresses for accessing data in RAM or other memory locations.
- **Special-Purpose Registers:** CPU architectures may include special-purpose registers for tasks like program counter (PC), stack pointer (SP), and status flags (e.g., carry, zero, overflow).
- **Register Renaming:** Some modern architectures employ techniques like register renaming to optimize instruction execution.
- **Context Switching:** Registers play a crucial role in context switching when the CPU switches between executing different tasks.
- **Performance Impact:** The organization of registers affects CPU performance, including instruction execution speed and efficiency.
- **CPU Architecture:** Different CPU architectures (e.g., x86, ARM, MIPS) have varying register organizations, influencing their capabilities and efficiency.

The specific register organization can vary greatly between different CPU architectures and designs, but these features collectively contribute to a processor's functionality and performance.

Advantages of General Register Organization:

- The efficiency of the CPU improves as the number of registers used in this organization grows.
- Because the instructions are encoded compactly, less memory space is required to hold the programme.

Disadvantages of General Register Organization:

- It's important to avoid using registers in ways that aren't essential. As a result, compilers must be more sophisticated in this regard.
- The usage of a high number of registers necessitates additional costs in the organization.

General register CPU organization of two types:

1. **Register-memory reference architecture (CPU with less register)**: In this organization Source 1 is always required in the register, source 2 can be present either in the register or in memory. Here two address instruction formats are compatible instruction formats.
2. **Register-register reference architecture (CPU with more register):** In this organization, ALU operations are performed only on registered data. So, operands are required in the register. After manipulation, the result is also placed in a register. Here three address instruction formats are the compatible instruction format.

Purpose of a General Register Organization:

A general register organization (GRO) is like a rulebook for a company or group. It specifies precise regulations that everyone must follow to maintain things operating smoothly. It's important because it helps everyone track the proper management, especially when there are laws to obey. Having a GRO puts all the critical details in one spot, simplifies verifying, and ensures accuracy. It is helpful when outside people check everything, like during an audit. People's roles, money transactions, and how things are done are all considered when making a GRO. GRO makes things run better, saves money, and works well for any group.

Single Accumulator Organization:

In an accumulator type organization, all the operations are performed with an implied accumulator register.

The instruction format in this type of computer uses one address field.

For example, the instruction that specifies an arithmetic addition defined by an assembly language instruction as

ADD X

Where X is the address of the operand. The ADD instruction in this case results in the operation AC ← AC + M[X]. AC is the accumulator register and M[X] symbolizes the memory word located at address X.

Stack Organization:

Introduction:

A stack is a data storage structure in which the most recent thing deposited is the most recent item retrieved. It is based on the LIFO concept (Last-in-first-out). The stack is a collection of memory locations containing a register that stores the top-of-element address in digital computers.

Stack's operations are:

- **Push:** Adds an item to the top of the stack
- **Pop:** Removes one item from the stack's top

What is Stack Organization?

The Last In First Out (LIFO) list is another name for stack. It is the CPU's most crucial feature. It saves information so that the last element saved is retrieved first. A memory space with an address register is called a stack. This register, known as the Stack Pointer, affects the stack's address (SP). The address of the element at the top of the stack is continuously influenced by the stack pointer.

Implementation of Stack:

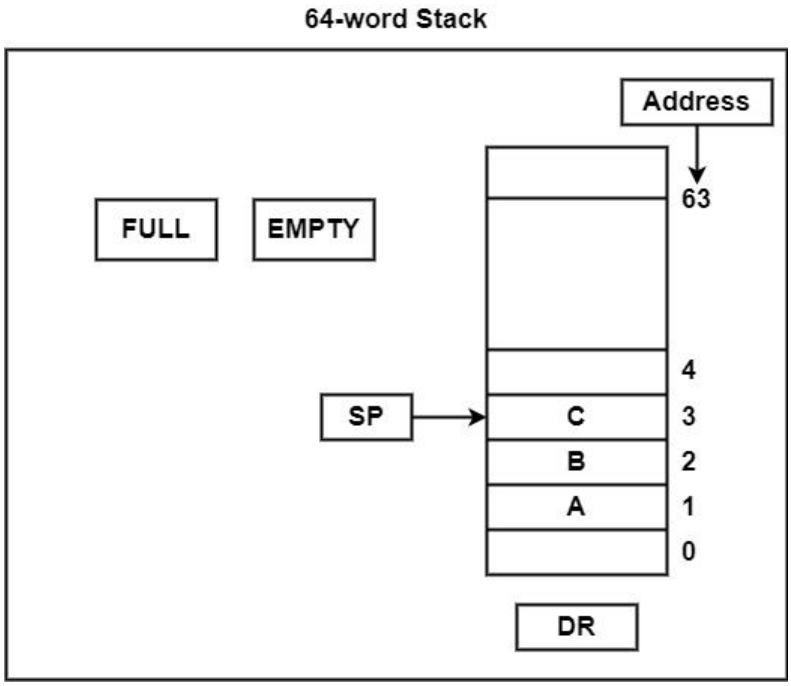
The stack can be implemented using two ways:

- Register Stack
- Memory Stack

1. Register Stack:

A stack of memory words or registers may be placed on top of each other. Consider a 64-word register stack like the one shown in the diagram. A binary number, which is the address of the element at the top of the stack, is stored in the stack pointer register. The stack has the three elements A, B, and C.

The stack pointer holds C's address, which is 3. C is at the top of the stack. The top element is removed from the stack by reading the memory word at address 3 and decreasing the stack pointer by one. As a result, B is at the top of the stack, and the SP is aware of B's address, which is 2. It may add a new word to the stack by increasing the stack pointer and inserting a word in the newly increased location.



The stack pointer includes 6 bits, because $2^6 = 64$, and the SP cannot exceed 63 (111111 in binary). After all, if 63 is incremented by 1, therefore the result is 0(111111 + 1 = 1000000). SP holds only the six least significant bits. If 000000 is decremented by 1 thus the result is 111111.

Therefore, when the stack is full, the one-bit register 'FULL' is set to 1. If the stack is null, then the one-bit register 'EMTY' is set to 1. The data register DR holds the binary information which is composed into or readout of the stack.

First, the SP is set to 0, EMTY is set to 1, and FULL is set to 0. Now, as the stack is not full (FULL = 0), a new element is inserted using the push operation.

The push operation is executed as follows –

```
SP←SP + 1    // increments the stack pointer
K[SP] ← DR   // writes the element on the top of the stack
If (SP = 0) then (FULL ← 1) // to check if the stack is full
EMTY ← 0  // to mark that stack is not empty
```

The stack pointer is incremented by 1 and the address of the next higher word is saved in the SP. The word from DR is inserted into the stack using the memory write operation. The first element is saved at address 1 and the final element is saved at address 0. If the stack pointer is at 0, then the stack is full and 'FULL' is set to 1. This is the condition when the SP was in location 63 and after incrementing SP, the final element is saved at address 0. During an element is saved at address 0, there are no more empty registers in the stack. The stack is full and the 'EMTY' is set to 0.

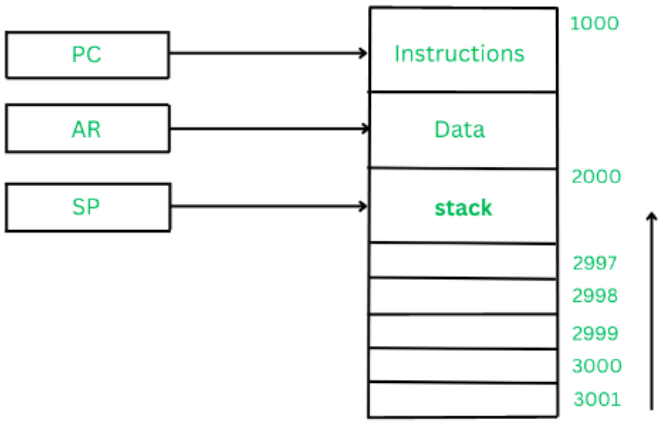
A new element is deleted from the stack if the stack is not empty (if EMTY = 0). The pop operation includes the following sequence of micro-operations −

```
DR←K[SP]    // to read an element from the top of the stack
SP ← SP - 1   // to decrement the stack pointer
If (SP = 0) then (EMTY ← 1)   // to check if the stack is empty
FULL ← 0  // to mark that stack is not full
```

The top element from the stack is read and transfer to DR and thus the stack pointer is decremented. If the stack pointer reaches 0, then the stack is empty and 'EMTY' is set to 1. This is the condition when the element in location 1 is read out and the SP is decremented by 1.

2. Memory Stack:

A stack may be implemented in a computer's random access memory (RAM). A stack is implemented in the CPU by allocating a chunk of memory to a stack operation and utilizing a processor register as a stack pointer. The stack pointer is a CPU register that specifies the stack's initial memory address.



Memory Stack Organization

As we can see in the figure, these three registers are connected to a **common address bus** and either one of them can provide an address for memory.

Stack Pointer is first going to point at the address 3001, and then the stack will grow with the **decreasing addresses**. It means that the first item is going to be stored at address 3001, the second item at address 3000, and the items can keep getting stored in the stack until it reaches the last address 2000 where the last item will be held.

Here the data which is getting inserted into the Stack is obtained from the **Data Register** and the data retrieved from the Stack is also read by the Data Register.

Now, let's see the working of **PUSH** and **POP** operations in Memory Stack Organization -

PUSH:

This operation is used to insert a new data item into the top of the Stack. The new item can be inserted as follows –

SP ←SP-1
M[SP]← DR

In the first step, the Stack Pointer is **decremented** to point at the address where the data item will be stored.

Then, by using the memory write operation, the data item from Data Register gets inserted into the top of the stack (at the address where the Stack Pointer is pointing).

POP:

This operation is used to delete a data item from the top of the Stack. Data item can be deleted as follows –

DR←M[SP]
SP←SP+1

In the first step, the top data item is read from the Stack into the Data Register. The Stack Pointer is then **incremented** to point at the next data item in the stack. Push or Pop operations can be performed with the help of the following microoperations:

- Access to memory with the help of Stack Pointer (SP), and
- Updating the stack.

It totally depends upon the organization of the stack whether the Stack Pointer (SP) is updated by incrementing or decrementing the address values.

In this case, the Stack Pointer grows by decreasing the memory address. The Stack may be made in a way that the Stack Pointer grows by increasing the memory also.

Since the address is always available and automatically updated in the Stack Pointer, the CPU can refer to the Memory Stack without having to specify an address.

Reverse Polish Notation in Stack:

The reverse polish notation in the stack is also known as **postfix expression**. Here, we use stack to solve the postfix expression.

From the postfix expression, when some operand is found, we push it into the stack, and when some operator is found, we pop elements from the stack, and after that, the operation is performed in the correct sequence, and the result is also stored in the stack.

For example, we are given this expression in the form of an array,

["18", "5", "*", "6", "/"]

From the given array we can deduce expression as,

((18 * 5) / 6) = 15

Approach:

1. We will access all the elements of the array one by one. While accessing all the elements, we will check if the current element is matching with the special character ('+', '*', '/', '-').
2. If the element does not match with any of the given special characters, then we will push that element into the stack.
3. After that, if any element is matched with the special character, we will pop the first two elements from the stack and we will perform the action and then push the result obtained by the operation into the stack again.
4. We will perform steps 1, 2, 3 for every array element.
5. In the end, we will pop the final result from the stack and will print the result.

Advantages of Stack Organization:

- Complex arithmetic statements may be rapidly calculated
- Instruction execution is rapid because operand data is stored in consecutive memory areas
- The instructions are minimal since they don't contain an address field

Disadvantages of Stack Organization:

- The size of the program increases when we use a stack
- It's in memory, and memory is slower in several ways than CPU registers. It generally has a lesser bandwidth and a longer latency. Memory accesses are more difficult to accelerate

Instruction Format:

The instruction formats are a sequence of bits (0 and 1). These bits, when grouped, are known as fields. Each field of the machine provides specific information to the CPU related to the operation and location of the data.

The instruction format also defines the layout of the bits for an instruction. It can be of variable lengths with multiple numbers of addresses. These address fields in the instruction format vary as per the organization of the registers in the CPU. The formats supported by the CPU depend upon the Instructions Set Architecture implemented by the processor.

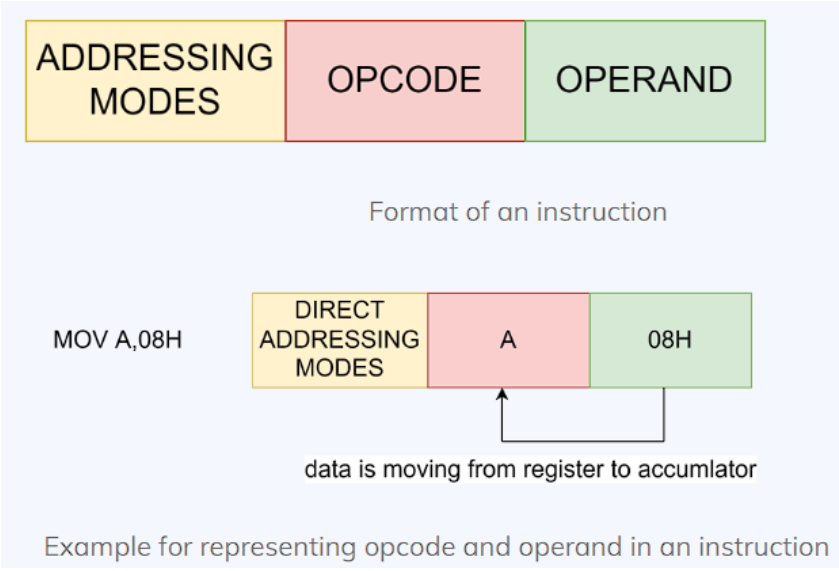
The operations specified by a computer instruction are executed on data stored in memory or processor registers. The operands residing in processor registers are specified with an address. The registered address is a binary number of k bits that defines one of the 2^k registers in the CPU. Thus, a CPU with 16 processors registers R0 through R15 and will have a four-bit register address field.

Example: The binary number 0011 will designate register R3.

Opcode and operand are the two main components of an instruction to execute. So, let's understand both of them one by one.

Opcode: Operation codes are known as opcodes. The first component of an instruction, known as an opcode, instructs the computer on what task to carry out. For each of its functions, a computer has an operation code or opcode. What to do with the variable or data written beside it is specified by an instruction that is given to the processor.

Operand: The second component of an instruction, known as an operand, instructs the computer where to locate or store the data or instructions. Different computers have different numbers of operands. Each instruction provides instructions on what to do and how to do it to the CPU's Control Unit. Depending on the issue the computers are given, the operations include arithmetic, logic, branching, etc.



What are the Different Types of Filed in Instruction?

A computer performs a task based on the instruction provided. Instruction in computers comprises groups called fields. These fields contain different information for computers everything is in 0 and 1 so each field has different significance based on which a CPU decides what to perform. The most common fields are:

- The operation field specifies the operation to be performed like addition.
- Address field which contains the location of the operand, i.e., register or memory location.
- Mode field which specifies how operand is to be founded.

Instruction is of variable length depending upon the number of addresses it contains. Generally, CPU organization is of three types based on the number of address fields:

- Single Accumulator organization
- General register organization
- Stack organization

In the first organization, the operation is done involving a special register called the accumulator. In the second multiple registers are used for the computation purpose. In the third organization the work on stack basis operation due to which it does not contain any address field. Only a single organization doesn't need to be applied, a blend of various organizations is mostly what we see generally.

Types of Instructions:

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

1. Zero Address Instruction:

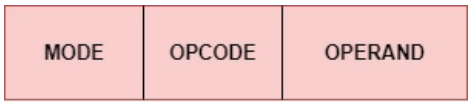
This instruction does not have an operand field, and the location of operands is implicitly represented. The stack-organized computer system supports these instructions. To evaluate the arithmetic expression, it is required to convert it into reverse polish notation.

Example: Consider the below operations, which shows how X = (A + B) ∗ (C + D) expression will be written for a stack-organized computer.

```
TOS: Top of the Stack
PUSH    A    TOS ← A
PUSH    B    TOS ← B
ADD          TOS ← (A + B)
PUSH    C    TOS ← C
PUSH    D    TOS ← D
ADD          TOS ← (C + D)
MUL          TOS ← (C + D) * (A + B)
POP     X    M [X] ← TOS
```

2. One Address Instruction:

This instruction uses an implied accumulator for data manipulation operations. An accumulator is a register used by the CPU to perform logical operations. In one address instruction, the accumulator is implied, and hence, it does not require an explicit reference. For multiplication and division, there is a need for a second register. However, here we will neglect the second register and assume that the accumulator contains the result of all the operations.



Example: The program to evaluate X = (A + B) * (C + D) is as follows:

```
LOAD    A    AC ← M [A]
ADD     B    AC ← A [C] + 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
```

All operations are done between the accumulator (AC) register and a memory operand.

M[ ] is any memory location.

M[T] addresses a temporary memory location for storing the intermediate result.

This instruction format has only one operand field. This address field uses two special instructions to perform data transfer, namely:

- **LOAD:** This is used to transfer the data to the accumulator.
- **STORE:** This is used to move the data from the accumulator to the memory.

3. 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.

Example: The program to evaluate X = (A + B) ∗ (C + D) is as follows:

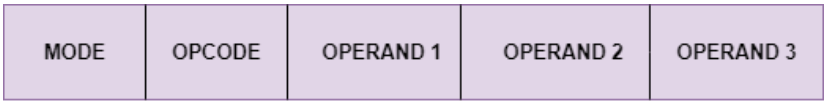
```
MOV     R1, A       R1 ← M [A]
ADD     R1, B       R1 ← R1 + M [B]
MOV     R2, C       R2 ← M [C]
ADD     R2, D       R2 ← R2 + M [D]
MUL     R1, R2      R1 ← R1*R2
MOV     X, R1       M [X] ← R1
```

The MOV instruction transfers the operands to the memory from the processor registers. R1, R2 registers.

4. Three Address Instruction:

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.



Example: The program in assembly language X = (A + B) ∗ (C + D) Consider the instructions given below that explain each instruction's register transfer operation.

```
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
```

Two processor registers, R1 and R2.

The symbol M [A] denotes the operand at memory address symbolized by A. The operand1 and operand2 contain the data or address that the CPU will operate. Operand 3 contains the result's address.

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.

Overall, the choice of instruction format depends on the specific requirements of the computer architecture and the trade-offs between code size, execution time, and flexibility.

Instruction Type	**Number of Addresses**	**Example**	**Description**
Single-Address	1	**LOAD A**	Specifies one memory address. Operand is implicitly or explicitly at that address.
Two-Address	2	**ADD A, B**	Specifies two addresses: one for the source operand (A), and one for the destination (B).
Three-Address	3	**ADD C, A, B**	Specifies three addresses: one for each operand (A, B), and one for the result (C).
Zero-Address	0	**ADD**	Operands and result are implicitly on top of the stack or specified by the top of the stack.

Addressing Modes:

The Addressing mode refers to how the operand of an instruction is specified. It specifies a rule for interpreting/modifying the address field of the instruction before referencing the operand.

A set of low-level instructions has operands and opcodes. An addressing mode has no relation with the opcode part. It basically focuses on presenting the address of the operand in the instructions.

The operands needed for the operation are located using the mode field. It is not necessary to have an address field in the instruction. An address field can contain either a memory address or a processor register if there is an address field.

Effective Address (EA):

The effective address refers to the address of an exact memory location in which an operand's value is actually present.

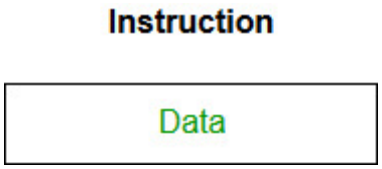
Types of Addressing Modes:

The different ways of specifying the location of an operand in an instruction are called as 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. Displacement Addressing Mode
9. Relative Addressing Mode
10. Indexed Addressing Mode
11. Base Register Addressing Mode
12. Auto-Increment Addressing Mode
13. Auto-Decrement Addressing Mode

1. Implied/ Implicit Addressing Mode:

The operands are implicitly specified in the instruction's definition. All register reference the instructions that use an accumulator are implied-mode instructions. Zero-address instructions in a stack-organized computer are also implied-mode instructions because the operands are implied to be on top of the stack.



Example: CLC (used to reset Carry flag to 0)

2. Stack Addressing Mode:

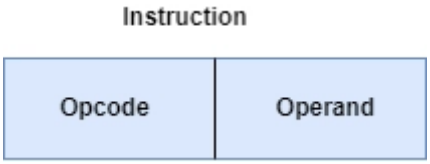
In this addressing mode, the operand is contained at the top of the stack.

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:

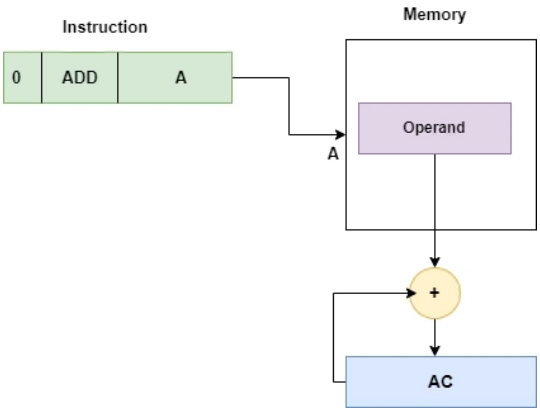
The operand is defined in the instruction itself. This mode instruction has an operand field instead of an address field. The operand field contains the actual operand used with the specified operation in the instruction. The immediate-mode instructions help initialize registers to a constant value.



For example, ADD 8 will increment the value stored in the accumulator by 8.

4. Direct Addressing Mode:

The effective address of the operand resides in the address field of the instruction. The operand resides in the memory, and the address field of the instruction gives its address. Only one reference to the memory is required to fetch the operand, and no additional calculations need to be done to find the effective address of the operand. It is also known as **absolute addressing mode**.

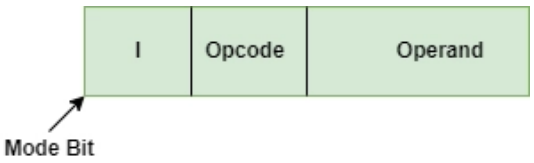


Example: **MOV A, [2000]** (Move the contents of memory location 2000 to register A).

5. Indirect Addressing Mode:

The address field of the instruction gives the address of the memory location that contains the effective address of the operand. Two references to the memory are required to fetch the operand: The control fetches the instruction from memory and uses

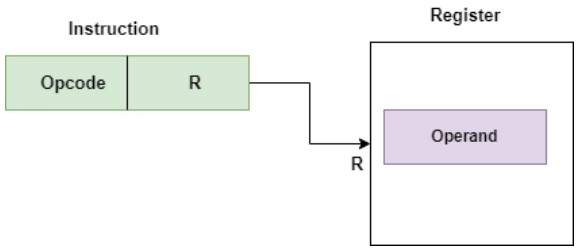
its address part to reaccess the memory that stores the effective address. This addressing mode slows down the execution as it requires multiple memory lookups to find the operand.



Example: **MOV A, [B]** (Move the contents of the memory address stored in register B to register A).

6. Register Direct Addressing Mode:

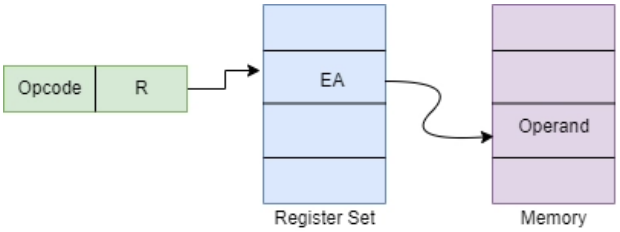
The operands that reside within the CPU are stored in the registers. The specific register is selected from a register field in the instruction. No reference to the memory is required to fetch the operand. The only difference between the Direct addressing mode and the register direct addressing mode is that the instruction address field refers to a CPU register instead of the main memory.



Example: **ADD A, B** (Add the contents of registers A and B).

7. Register Indirect Addressing Mode:

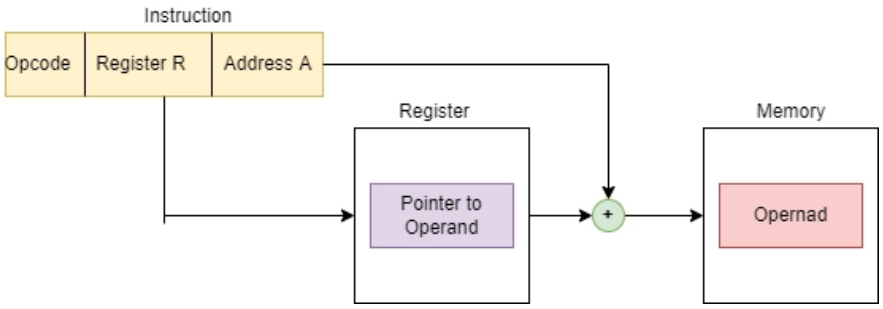
The instruction defines a register in the CPU that stores the effective address of the operand in memory. Only one reference to the memory is required to fetch the operand. The specified register contains the address of the operand instead of the operand. The only difference between the Indirect addressing mode and the register indirect addressing mode is that the instruction address field refers to a CPU register.



For example: ADD R1, R2 (Here, the content of R2 is added to R1. R1 R2 represents registers).

8. Displacement Addressing Mode:

The indexed register content is added to the instruction's address part to obtain the effective address of the operand.



EA = A + (R)

Here, the address field holds two values, A: Base value R: displacement value.

9. Relative Addressing Mode:

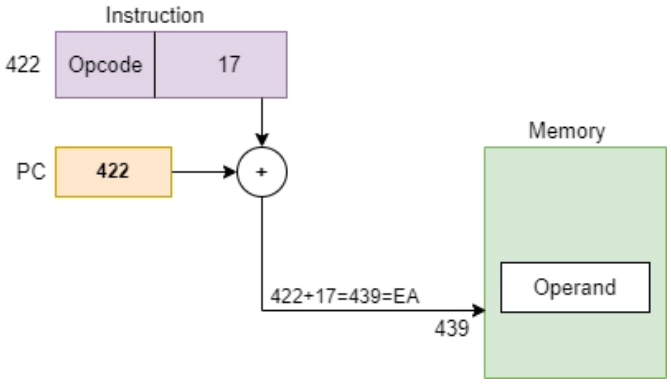
This mode is another version of the displacement address mode. The program counter's content is added to the instruction's address part to obtain the effective address.

EA = A + (PC)

Here, EA: Effective address, PC: program counter.

The instruction's address part is usually a signed number that can be positive or negative. After the instruction's address is fetched, the value of the program counter increases immediately, irrespective of whether the fetched instruction has been executed or not. PC: It contains the address of the next instruction to be executed.

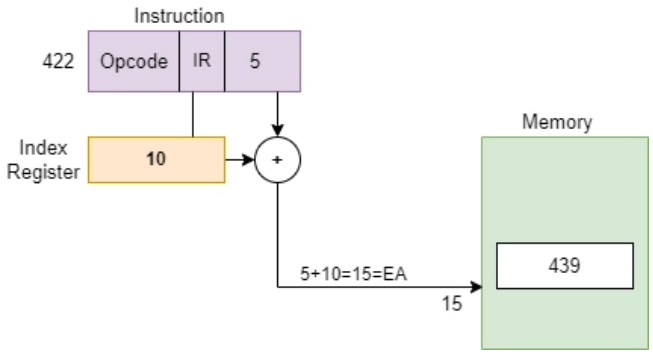
Consider an example:



The program counter contains the number 422, and the address part of the instruction contains the number 17. The instruction at location 421 is read during the fetch phase, and the program counter is then incremented by one to 422 + 17 = 439.

10. Indexed Addressing Mode:

The index register's content is added to the instruction's address to obtain the effective address.
EA = content of index register + Instruction address part



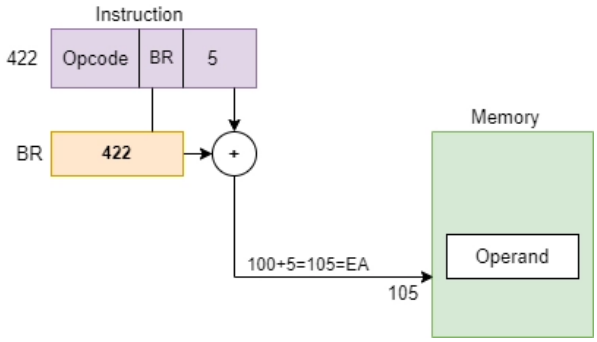
Example: **MOV A, [SI + 10]** (Move the contents of the memory address at (SI + 10) to register A).

11. Base Register Addressing Mode:

This mode is another version of the displacement address mode. To obtain the effective address, the base register's content is added to the instruction's address.

EA = A + (R)

A: Displacement, R: Pointer to the base address.



Example: **MOV A, [BX + 20]** (Move the contents of the memory address at (BX + 20) to register A).

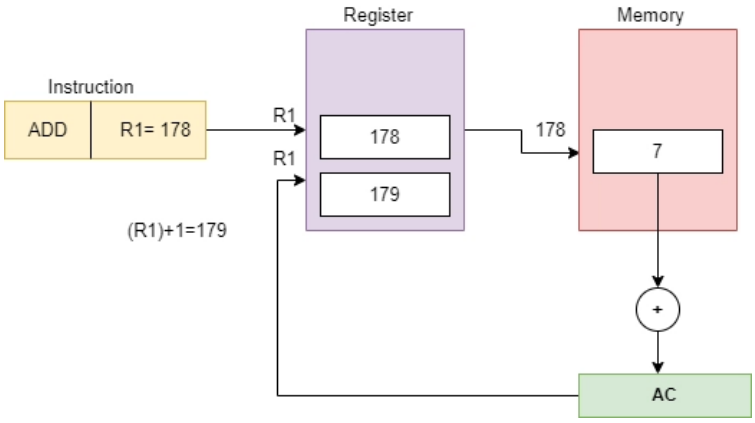
12. Auto-Increment Addressing Mode:

This mode is a special Register Indirect Addressing Mode case. The register is incremented/decremented after or before its value is utilized.

EA = content of the register.

The content of the register is increment automatically by step size ‘d’ after accessing the operand, where the step size ‘d’ depends on the size of the accessed operand. Only one reference memory is required to fetch the operand.

Consider an example:



Here, the Effective Address (R) =178, and the operand in AC are 7. After loading R1 is incremented by 1 and becomes 179.

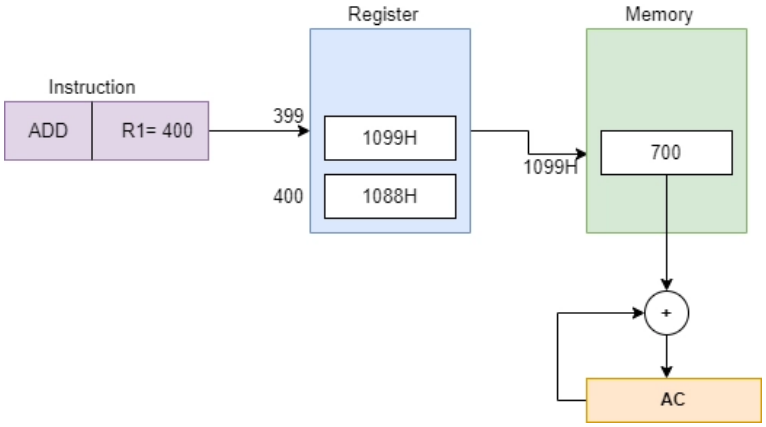
13. Auto-Decrement Addressing Mode:

This mode is also a special Register Indirect Addressing Mode case.

EA = content of the register - step size.

The content of the register is decremented by step size ‘d’ after accessing the operand, where the step size ‘d’ depends on the size of the accessed operand. Only one reference memory is required to fetch the operand.

Consider an example:



Here, register R1 is decremented by 1 (400-1=399) prior to the instruction execution which implies that the operand loaded to the AC is of address 1099H instead of 1088H. Hence, the Effective Address is 1099H.

Applications of Addressing Modes:

- **Immediate Addressing Mode:** It initializes the register to a constant value.
- **Direct Addressing Modes and Register Direct Addressing Mode:** It helps access static data and implement variables.
- **Indirect Addressing Modes and Register Indirect Addressing Mode:** It helps implement pointers and pass arrays as parameters.
- **Relative Addressing Mode:** It helps in program relocation at runtime. And in changing the sequence of instructions during execution.
- **Base Register Addressing Mode:** It helps in writing relocatable code and handling recursive procedures.
- **Index Addressing Mode:** It helps in the array and record implementation.
- **Auto-Increment Addressing Mode and Auto-Decrement Addressing Mode:** It helps implements loops and stacks.

Advantages of Addressing Modes:

- They improve performance by efficiently utilizing the CPU cache and reducing the memory read latency
- Addressing Modes are used for implementing complex data structures in memory as they provide mechanisms such as indexing
- Program sizes can be reduced drastically as the code can be compacted which allows faster execution of instructions
- Addressing modes give you the flexibility to use different ways of specifying the address of operands in your instructions

Disadvantages of Addressing Modes:

- **Complexity:** Particularly for inexperienced programmers, some addressing modes might be challenging to comprehend and use. Due to its intricacy, code may include errors or be inefficient
- **Limited Flexibility:** Some addressing modes may limit the kind of operand operations that can be carried out. For instance, Immediate Mode restricts dynamic calculations to preset values only
- **Increased Memory Access:** Because Direct and Indirect modes need more memory accesses to retrieve operands, program performance may be slowed down
- **Register Dependency:** The availability of registers is very important for register modes. This can cause conflicts and inefficient resource utilization in systems with a finite number of registers
- **Code Size:** Certain addressing modes could provide longer and more complicated instructions, increasing the code size

Numerical Example:

To show the differences between the various modes, we will show the effect of the addressing modes on the instruction defined in Fig. 8-7.

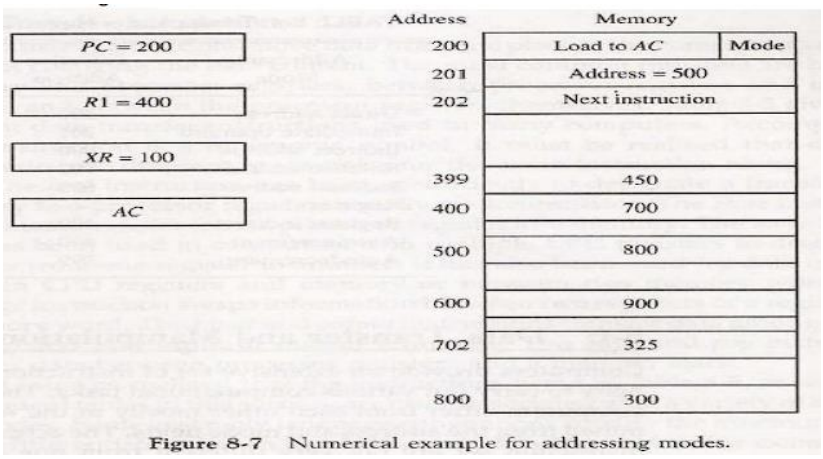


Figure 8-7 Numerical example for addressing modes.

The two-word instruction at address 200 and 201 is a "load to AC" instruction with an address field equal to 500.

The first word of the instruction specifies the operation code and mode, and the second word specifies the address part.

PC has the value 200 for fetching this instruction. The content of processor register R1 is 400, and the content of an index register XR is 100.

AC receives the operand after the instruction is executed.

- In the **direct address mode**, the effective address is the address part of the instruction 500 and the operand to be loaded into AC is 500.
- In the **immediate mode** the second word of the instruction is taken as the operand rather than an address, so 500 is loaded into AC.
- In the **indirect mode** the effective address is stored in memory at address 500. Therefore, the effective address is 800 and the operand is 300.
- In the **relative mode** the effective address is 500 + 202 =702 and the operand is 325. (The value in PC after the fetch phase and during the execute phase is 202.)
- In the **index mode** the effective address is XR+ 500 = 100 + 500 = 600 and the operand is 900.
- In the **register mode** the operand is in R1 and 400 is loaded into AC.
- In the **register indirect mode**, the effective address is 400, equal to the content of R1 and the operand loaded into AC is 700.
- The **auto-increment mode** is the same as the register indirect mode except that R1 is incremented to 401 after the execution of the instruction.
- The **auto-decrement mode** decrements R1 to 399 prior to the execution of the instruction. The operand loaded into AC is now 450.

Table 8-4 lists the values of the effective address and the operand loaded into AC for the nine addressing modes.

TABLE 8-4 Tabular List of Numerical Example

Addressing Mode	Effective Address	Content of *AC*
Direct address	500	800
Immediate operand	201	500
Indirect address	800	300
Relative address	702	325
Indexed address	600	900
Register	—	400
Register indirect	400	700
Autoincrement	400	700
Autodecrement	399	450

Types of Instructions:

There are certain basic operations included in every computer's instructions set. The computer instructions are classified into three categories:

- Data Transfer Instructions
- Data Manipulation Instructions
- Program Control Instructions

Data Transfer Instructions:

Data transfer instructions move data from one location to another, without changing the binary information content. They are also called copy instructions.

Typically, the transfers are between memory and processor registers, between processor registers and input and output registers, and among the processor registers themselves.

Following is a table of data transfer instructions.

Mnemonic	**Definition**	**Syntax**
MOV	Move instruction copies the data from the source to the destination.	Syntax: MOV destination, source
POP	Pop instruction is used to get the data from the stack.	Syntax: POP CX
XCHG	Exchange instruction exchanges the contents of the source & the destination.	Syntax: XCHG destination, source
PUSH	Push instruction is used to push data into the stack.	Syntax: PUSH source
LAHF	Load AH from Flag register instruction will load the AH register with the content of the lower byte of the flag register.	Syntax: LAHF

IN	This Input instruction is used to transfer data from the input unit to the accumulator.	Syntax: IN accumulator, Port address
SAHF	Store AH to Flag register instruction stores the content of AH register to the lower byte of flag register.	Syntax: SAHF
OUT	Output instruction is used to transfer data from the accumulator to the output unit.	Syntax: OUT Port address, accumulator
LD	Load instruction will load the register that is defined in the instruction and the data segment (DS) from the source.	Syntax: LDS destination, source

Data Transfer operations:

The data transfer instructions move data b/w registers or b/w memory and registers.

Opcode	**Operand**	**Data transfer instructions.**	**Explanation**	**Example**
MOV	Rd, Rs	Move	Rd=Rs	MOV A, B
MOV	Rd, M	Move	Rd=Mc	MOV A, 2050
MOV	M, Rs	Move	M=Rs	MOV 2050, A
MVI	Rd, 8-bit data	Move Immediate	Rd=8-bit data	MVI A, 50
MVI	M, 8-bit data	Move Immediate	M=8-bit data	MVI 2050, 50
LDA	16-bit address	Load Accumulator Directly from Memory	A=contents at address	LDA 2050
STA	16-bit address	Store Accumulator Directly in Memory	Contents at address=A	STA 2050
LHLD	16-bit address	Load H & L Registers Directly from Memory	Directly loads at H & L register	LHLD 2050
LXI	r.p.,16-bit data	Load Register Pair with Immediate data	Indirectly loads at the accumulator A	LDAX H

In this table, R stands for register, M stands for memory, r.p. stands for register pair And an 'X' in the name of a data transfer instruction implies that it deals with a register pair (16-bits).

Advantages of Data Transfer Instructions:

A helpful method for managing data in a computer system is to employ data transfer instructions. They aid with swift and correct information transfer, enhancing the system's efficiency. The following are some benefits of following these guidelines:

- **Faster Operations:** Data transfers proceed considerably more quickly, saving time on chores like backups and migrations.
- **Less Memory Usage:** They utilise less memory, which frees up resources for other operations.
- **Improved System Performance:** Automation streamlines huge data transfers, enhancing system efficiency.
- **Improved Communication:** They facilitate improved resource utilisation by enhancing communication between various systems and applications.
- **Simple Big Data Handling:** They move massive amounts of data without manual labour, perfect for major processes involving several systems.
- **Accuracy:** Data transfers are more exact, preventing mistakes in transactions involving sensitive data or money.
- **Easier Coding:** Streamlining the coding procedure minimises complexity and shortens the time required to write code.

Disadvantages of Data Transfer Instructions:

Modern computer programming relies heavily on data transfer instructions to move data between various components. Although they facilitate quick and effective data transmission, they also have significant drawbacks. Let's examine these drawbacks:

- **Resource use:** Transferring data depletes memory and processing capacity, especially when working with huge amounts of data. The system's efficiency may suffer as a result.
- **Bandwidth restrictions:** Transferring vast volumes of data can be difficult or impossible if the bandwidth between components is constrained or unreliable. Timing problems also result in bottlenecks and slow the process down.
- **Security hazard:** Network congestion can prolong transmission times and raise the chance of sensitive data exposure. Problems with compressibility could prevent data compression from enhancing attack security.
- **Protocol incompatibility and instruction overhead:** When many protocols attempt to interact, compatibility problems may occur, slowing down performance. The administrative burden of handling data transfer instructions makes the procedure more complex.

Data Manipulation Instructions:

Data manipulation instructions are those instructions that manipulate or change the content of the data/registers/memory. It performs operations on data and provides the computational capabilities of the computer.

Data manipulation instructions can be categorized into three parts:

1) Arithmetic instruction

2) Logical and bit manipulation instructions

3) Shift instructions

1) Arithmetic Instruction:

Arithmetic instructions include increment, decrement, add, subtract, multiply, divide, add with Carry, subtract with Borrow, negate that is (2's) two's complement. If there's a negative number, it is considered as negate (so two's complement).

Generally, most computers carry instructions for all four of these operations. If computers have only addition (ADD) and possibly subtraction(SUB) instructions, the other two operations, i.e. multiplication(MUL) and division(DIV) must be generated using software subroutines. These four basic arithmetic operations are sufficient for solving scientific problems when expressed in numerical analysis methods.

The table given below shows the Arithmetic Instructions:

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

2) Logical and Bit manipulation Instruction:

We are having another list of instructions that is logical and bit manipulation instructions starting with clear (that means clear the content of accumulator), complement the accumulator, AND, OR, Exclusive-OR, Clear carry, Set carry, Complement carry, Enable interrupts, Disable interrupts, all these are logical and bit manipulation instructions.

These logical instructions consider each operand bit individually and treat it as a Boolean variable. Basically, logical instructions help perform binary operations on strings of bits stored in registers.

- Clear instruction means making all the bits of a register '0'.
- AND instruction is sometimes referred to as bit clear instruction or mask.
- OR instruction is sometimes referred to as bit set instruction.
- Set instruction means making all the bits of a register '1'.
- XOR instruction is referred to as bit complement instruction.

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

3) Shift Instructions:

Shift instructions allow the bits of a memory byte or register to be shifted one-bit place to the right or the Left.

There are basically two types of shift instructions — arithmetic and logical. Arithmetic shifts consider the contents of the memory byte or register to be a signed number. So, when the shift is made, the number is arithmetically divided by two (right shift) or multiplied by two (left shift). Logical shifts consider the contents of the register or memory byte to be just a bit pattern when the shift is made.

- **OP** is opcode field
- **RL** (It tells whether to shift it right or left).
- **REG (**It determines which register is to be shifted).
- **COUNT** (It tells the number of places to be shifted).
- **TYPE(** It tells the type of shifting from the list given below).

In right-shift operations, zeros are shifted into high-order vacated positions. And in the case of the left-shift operation, shifts the zero into low-order vacated positions.

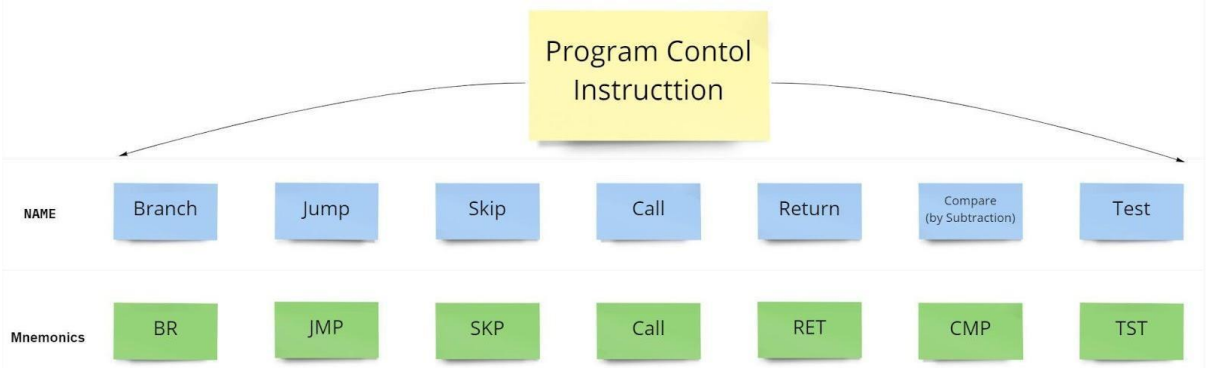
Name	Mnemonic
Logical shift right	SHR
Logical shift left	SHL
Arithmetic shift right	SHRA
Arithmetic shift left	SHLA
Rotate right	ROR
Rotate left	ROL
Rotate right through carry	RORC
Rotate left through carry	ROLC

Program Control Instructions:

Program control instructions modify or change the flow of a program. It is the instruction that alters the sequence of the program's execution, which means it changes the value of the program counter, due to which the execution of the program changes.

Features:

- These instructions cause a change in the sequence of the execution of the instruction.
- This change can be through a condition or sometimes unconditional.
- Flags represent the conditions.
- Flag-Control Instructions.
- Control Flow and the Jump Instructions include jumps, calls, returns, interrupts, and machine control instructions.
- Subroutine and Subroutine-Handling Instructions.
- Loop and Loop-Handling Instructions.



Program Control Instructions	Description
Branch (BR)	Branch which means it is an unconditional jump. It is unconditional branching wherever we specify the address we need to branch.
Skip (SKP)	Skip instructions is used to skip one(next) instruction. It can be conditional or unconditional. It does not need an address field. In the case of conditional skip instruction, the combination of conditional skip and an unconditional branch can be used as a replacement for the conditional branch.
Jump (JMP)	The jump instruction transfers the program sequence to the memory address given in the operand based on the specified flag.
Compare (CMP)	The Compare instruction performs a comparison via a subtraction, with difference not retained. CMP compares register sized values, with one exception.
CALL and RETURN	The CALL and RETURN instructions interrupt the flow of a program by passing control to an internal or external subroutine. An external subroutine is another program. The RETURN instruction returns control from a subroutine back to the calling program and optionally returns a value.
TEST	TEST instructions perform the AND of two operands without retaining the result, and so on.

Now let us move further, these all are unconditional jumps or unconditional branching, which means there is no certain condition based on which they can jump or alter the execution sequence.

Status Bit Conditions:

To check different conditions for branching instructions like CMP (compare) or TEST can be used. Certain status bit conditions are set as a result of these operations.

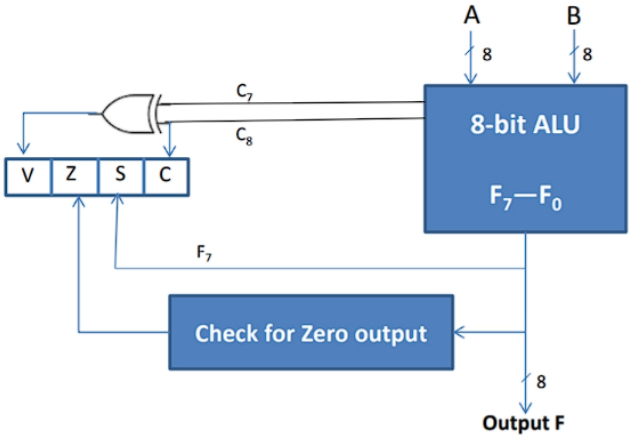
V	Z	S	C

Status bits mean that the value will be either 0 or 1 as it is a bit. We have four status bits:

- "V" stands for Overflow
- "Z" stands for Zero
- "S" stands for the Sign bit
- "C" stands for Carry.

Now, these will be set or reset based on the ALU (Arithmetic Logic Unit) operation carried out into the CPU. Let us discuss these bits before understanding the operation.

- Overflow(V) is based on certain bits, i.e., if extra bits are generated into our operation. Then we have Zero (Z).
- If the output of the ALU (Arithmetic Logic Unit) is 0, then the Z flag is set to 1, otherwise, it is set to 0.
- If the number is positive, the Sign(S) flag is 0, and if the number is negative, the Sign flag is 1.
- We have Carry(C), if the output of the thirst ALU operation generates Carry, then C is set to 1, else C is set to 0.



Let's see how these flags are affected. You can see in the figure this is an 8-bit ALU that performs arithmetic or logic operations on our data. Suppose we have two operands A and B of 8-bits on which we are performing certain arithmetic or logic operations. Arithmetic operation-addition is performed on A and B. We know that an extra Carry bit may be generated, which means eight Carry bits are generated. If the addition operation is performed on operands A and B, then the carry bits C0 to C7 may be generated. But we know that extra Carry may also be generated, which we term as C8. If C8 is generated, i.e., Carry is generated, reflecting our Carry flag, which results in the C flag or C status bit being set to 1.

Let's move further to the last two carries, C7 and C8. If the XOR of these two carries comes out to be 1, then we can say that the Overflow condition has happened and the V flag is set to 1; otherwise, it's set to 0. This is the most occurring case when we have negative numbers represented in 2's complement form.

Now moving on to the next flag which is the sign flag(S), the sign flag is set to 1 or 0 based on the output of the 8 bit ALU. As we know that if the number is positive, then the most significant bit of a number is represented to be 0, which means if F7 is 0 then we can say that the number is positive and if F7 is 1 then we say that the number is negative.

And lastly, we have a Zero flag, Zero status bit Z, "Z" is said to 1 if the output of all the bits from F0 to F7 is 0, then we can say that the zero flag is SET.

Now all these four bits that are "V", "Z", "S" and "C" are reflected based on the arithmetic or logic operation carried out on the 8 bit ALU.

Conditional Branch Instructions:

A conditional branch instruction is basically used to examine the values that are stored in the condition code register to examine whether the specific condition exists and to branch if it does.

Conditional branch instructions such as ‘branch if zero’ or ‘branch if positive’ specify the condition to transfer the execution flow. The branch address will be loaded in the program counter when the condition is met.

Each conditional branch instruction tests for a different combination of Status bits for a condition.

Mnemonic	Branch Condion	Tested Condition
BZ	Branch if Zero	Z=1
BNZ	Branch if not Zero	Z=0
BC	Branch if carry	C=1
BNC	Branch if no carry	C=0
BP	Branch if plus	S=0
BM	Branch if minus	S=1
BV	Branch if overflow	V=1
BNV	Branch if no overflow	V=0
Unsigned compare conditions (A-B)		
BHI	Branch if higher	A>B
BHE	Branch if higher or equal	A>=B
BLO	Branch if lower	A<B
BLOE	Branch if lower or Equal	A<=B
BE	Branch iS Equal	A=B
BNE	Branch is not equal	A≠B
Signed compare conditions(A-B)		
BGT	Branch if greater than	A>B
BGE	Branch if greater or equal	A>=B
BLT	Branch if less than	A<B
BLE	Branch if less or equal	A<=B
BE	Branch if equal	A=B
BNE	Branch if not equal	A≠B

Comparison Branch Instructions:

Remember A≥B is the complement of A<B and A≤B is the complement of A<B, which means if we know the condition of status bits for one, the condition for the other complementary relation is obtained by complement.

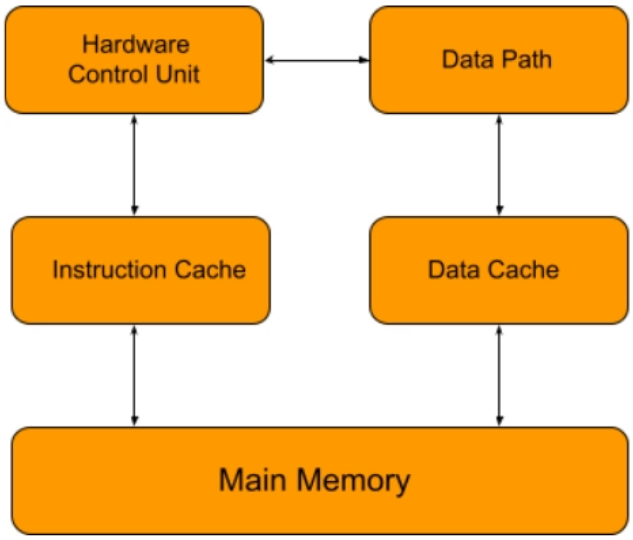
Mnemonic	Branch Condion	Tested Condition
BZ	Branch if Zero	Z=1
BNZ	Branch if not Zero	Z=0
BC	Branch if carry	C=1
BNC	Branch if no carry	C=0
BP	Branch if plus	S=0
BM	Branch if minus	S=1
BV	Branch if overflow	V=1
BNV	Branch if no overflow	V=0
Unsigned compare conditions (A-B)		
BHI	Branch if higher	A>B
BHE	Branch if higher or equal	A>=B
BLO	Branch if lower	A<B
BLOE	Branch if lower or Equal	A<=B
BE	Branch iS Equal	A=B
BNE	Branch is not equal	A≠B
Signed compare conditions(A-B)		
BGT	Branch if greater than	A>B
BGE	Branch if greater or equal	A>=B
BLT	Branch if less than	A<B
BLE	Branch if less or equal	A<=B
BE	Branch if equal	A=B
BNE	Branch if not equal	A≠B

Reduced Instruction Set Computer (RISC):

RISC is a microprocessor architecture that uses a simple set of instructions that can be substantially modified. It is designed to reduce the time it takes for instructions to execute by optimizing and reducing the number of instructions. It means that each instruction cycle has only one clock per cycle, and each cycle consists of three parameters: fetch, decode, and execute. The RISC processor can also combine multiple complex instructions into a simple one. RISC chips require several transistors, making them less expensive to develop and reducing instruction execution time.

Examples of RISC processors are PowerPC, Microchip PIC, SUN's SPARC, RISC-V.

RISC Architecture:



Features of RISC Processor:

Some of the crucial features of the RISC processor are -

1.) RISC processors use one clock per cycle (CPI) to execute each instruction in a computer. Each CPI also comprises the methods for fetching, decoding, and executing computer instructions.

2.) Multiple registers in RISC processors allow them to hold instructions, reply fast to the computer, and interact with computer memory as little as possible.

3.) The RISC processors use the pipelining technique to execute multiple parts or stages of instructions to perform more efficiently.

4.) RISC has a simple addressing mode and fixed instruction length for the pipeline execution.

5.) It uses LOAD and STORE instruction to access the memory location.

Advantages of RISC:

- **Simpler instructions:** RISC processors use a smaller set of simple instructions, which makes them easier to decode and execute quickly. This results in faster processing times.
- **Faster execution:** Because RISC processors have a simpler instruction set, they can execute instructions faster than CISC processors.
- **Lower power consumption:** RISC processors consume less power than CISC processors, making them ideal for portable devices.

Disadvantages of RISC:

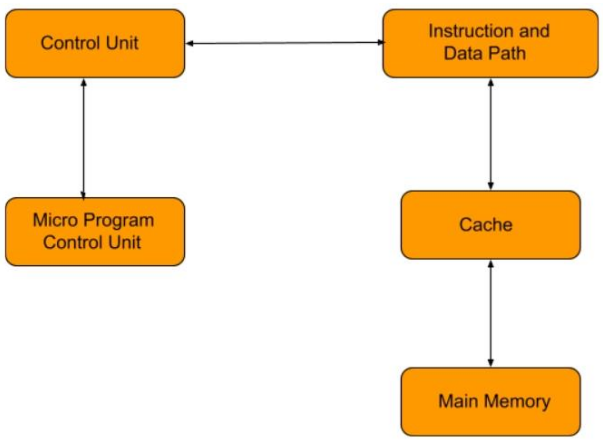
- **More instructions required:** RISC processors require more instructions to perform complex tasks than CISC processors.
- **Increased memory usage:** RISC processors require more memory to store the additional instructions needed to perform complex tasks.
- **Higher cost:** Developing and manufacturing RISC processors can be more expensive than CISC processors.

Complex Instruction Set Computer (CISC):

Intel developed the CISC processor. It has an extensive collection of complex instructions that range from simple to very complex and specializes in the assembly language level, which takes a long time to execute the instructions. So, CISC approaches reducing the number of instructions on each program and ignoring the number of cycles per instruction. It emphasizes building complex instructions directly in the hardware because the hardware is always faster than software. However, CISC chips are relatively slower than RISC chips but use little instructions.

Examples of CISC processors are AMD, Intel x86, and the System/360.

CISC Architecture:



Features of CISC Processor:

Some of the crucial features of the RISC processor are -

1.) CISC may take longer than a single clock cycle to execute the code.

2.) The length of the code is short, so it requires minimal RAM.

3.) It provides more accessible programming in assembly language.

4.) It focuses on creating instructions on hardware rather than software because they are faster to develop.

5.) It comprises fewer registers and more addressing nodes, typically 5 to 20.

Advantages of CISC:

- **Reduced code size:** CISC processors use complex instructions that can perform multiple operations, reducing the amount of code needed to perform a task.
- **More memory efficient:** Because CISC instructions are more complex, they require fewer instructions to perform complex tasks, which can result in more memory-efficient code.
- **Widely used:** CISC processors have been in use for a longer time than RISC processors, so they have a larger user base and more available software.

Disadvantages of CISC:

- **Slower execution:** CISC processors take longer to execute instructions because they have more complex instructions and need more time to decode them.
- **More complex design:** CISC processors have more complex instruction sets, which makes them more difficult to design and manufacture.
- **Higher power consumption:** CISC processors consume more power than RISC processors because of their more complex instruction sets.

CPU Performance:

Both approaches try to increase the CPU performance

- **RISC:** Reduce the cycles per instruction at the cost of the number of instructions per program.

$$CPU\ Time = \frac{Seconds}{Program} = \frac{Instructions}{Program} X \frac{Cycles}{Instructions} X \frac{Seconds}{Cycle}$$

- **CISC:** The CISC approach attempts to minimize the number of instructions per program but at the cost of an increase in the number of cycles per instruction.

Earlier when programming was done using assembly language, a need was felt to make instruction do more tasks because programming in assembly was tedious and error-prone due to which CISC architecture evolved but with the uprise of high-level language dependency on assembly reduced RISC architecture prevailed.

Example: Suppose we have to add two 8-bit numbers:

- **CISC approach:** There will be a single command or instruction for this like ADD which will perform the task.
- **RISC approach:** Here programmer will write the first load command to load data in registers then it will use a suitable operator and then it will store the result in the desired location.

So, add operation is divided into parts i.e. load, operate, store due to which RISC programs are longer and require more memory to get stored but require fewer transistors due to less complex command.

Difference Between RISC and CISC:

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