Programming Paradigm

Programming paradigms are fundamental style or approach to programming that guide how a programmer structures and writes code. It shapes how programmers think about problem solving, organize logic, and how they use language constructs and tools. Different programming paradigms offer various ways of conceptualizing and interacting with data and operations in a program.

Key Programming Paradigms

Programming paradigm can be classified into two Imperative and Declarative

Diagram

Description automatically generated

What is Imperative programming?

Direct
Efficient
Inflexible
Side Effects
 The programmer writes instructions that tell the computer what to accomplish and how to do it. The imperative programming code focuses on creating algorithms that tell the computer how to do things. It closely relates to how hardware works.

It involves specifying a sequence of commands for the computer to execute in order to achieve a desired outcome. It is also characterized by a clear sequence of instructions that change the state of the program as it progresses.

Imperative programming paradigm is concerned with **HOW** you’re actually going to get to the solution.

Types of Imperative Programming Paradigm

* Procedural
* Structural
* Object-Oriented Programming (OOP)

Declarative programming

focuses on what the program should accomplish, rather than detailing the steps to achieve it. The logic of the computation is described without necessarily specifying its control flow or the exact steps to be taken.  
Think of declarative programming like ordering a meal at a restaurant. You tell the waiter what you want to eat, but you don't need to explain how to cook the meal. The kitchen (or the computer, in this case) takes care of the details.

The declarative approach is more concerned with **WHAT** you want.

Types of Declarative programming

* Logic programming
* Database approach
* Functional programming

**Low level language -Assembly language**

Assembly language is not typically considered a programming paradigm itself. Instead, assembly language is a low-level programming language that represents a specific way of interacting with the hardware of a computer. It provides a human-readable representation of machine code instructions, which are directly executed by the computer's processor.

Assembly language is often associated with the imperative programming paradigm because it involves specifying a sequence of instructions that are executed one after the other. However, assembly language itself does not provide higher-level abstractions or concepts that characterize paradigms like procedural or object-oriented.

Assembly language is a specific to the architecture of a particular computer or processor. Assembly language is considered a bridge between machine code and higher-level programming languages.

Assembly language,

* instructions are written using mnemonics that represent specific operations or actions to be performed by the computer's processor. Each mnemonic corresponds to a specific machine code instruction that the processor can directly execute.
* allows for specifying memory addresses and data manipulation.
* a good understanding of the computer's architecture is needed when writing assembly language, including its registers, memory organization, and instruction set.
* often used in areas where performance is crucial, such as operating systems, device drivers, embedded systems, and real-time systems. It is also used in reverse engineering and low-level system programming.
* Each computer architecture has its own assembly language. Examples include x86 assembly language for Intel processors, ARM assembly language for ARM-based processors, and MIPS assembly language for MIPS processors. Assemblers are used to convert assembly language code into machine code that can be executed by the processor.

The structure of assembly language

The structure of assembly language and machine code instructions is the same.

Each instruction has an

* opcode that identifies the operation to be carried out by the CPU.
* an operand that identifies the data to be used by the opcode.

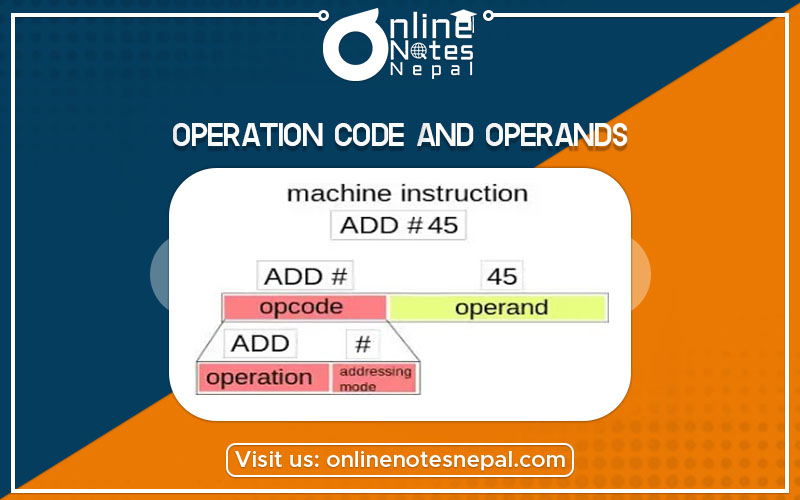
The opcode (operation code) is a part of the assembly language instruction that specifies the operation to be performed. It's a mnemonic representation of an underlying machine code instruction, making it easier for humans to read and write.

Examples of opcodes include MOV, ADD, SUB, JMP, and so on. Each opcode corresponds to a specific operation the CPU understands, like moving data, performing arithmetic, or changing the flow of control.

The operand

The operand(s) of an assembly instruction specify the data or memory address to be operated on or the data locations. The number and type of operands depend on the specific instruction. Operands can be registers, immediate values, memory addresses, or labels. Operands can be immediate values (like a constant number), CPU registers, or memory addresses.

A diagram of a graph

Description automatically generated with medium confidence

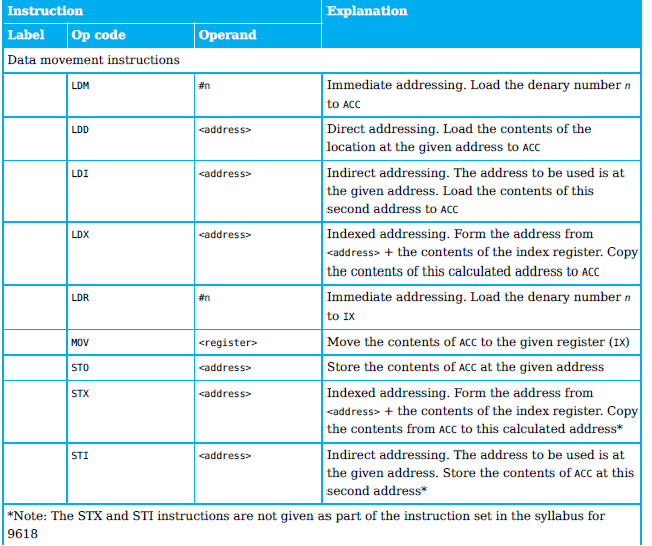
Assembly Language instruction set.

What is an instruction set?

An instruction set, often abbreviated as ISA (Instruction Set Architecture), is a collection of instructions that a processor can execute. It includes a variety of instructions that specify different operations, such as arithmetic and logical operations, data movement, control flow, and input/output operations.

Each instruction typically has an opcode that represents the specific operation to be performed, as well as the addressing mode that specifies how the operands (data or memory addresses) for the operation are accessed

Data Movement Instruction



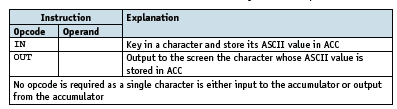
Arithmetic Operation

A white and blue box with black text

Description automatically generated

*Input and output of data instructions*

These instructions allow data to be read from the keyboard or output to the screen.



*Unconditional and conditional instructions*

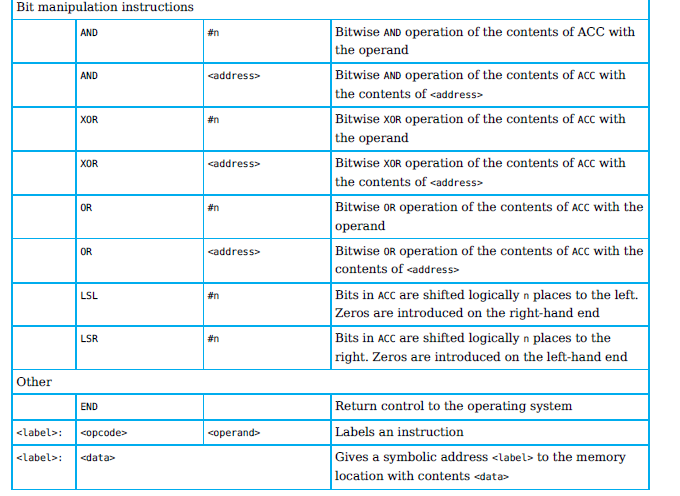
A white and blue text with black text

Description automatically generated

*Compare instructions*

A close-up of a list of information

Description automatically generated

Bit manipulation

Addressing modes

Addressing modes in assembly language are

* the ways in which operands are specified in instructions.
* It specifies how to access operands or data in memory or registers.
* Allow programmers to manipulate data efficiently and express complex operations.
* The specific set of addressing modes available depends on the architecture and instruction set of the processor.

**Direct /Absolute addressing –** the contents of the memory location in the operand are used.   
The operand's memory address is directly specified in the instruction. For instance, MOV AX, [2000H] means the value at memory address 2000H is moved into the AX register.

**Indirect addressing** – the contents of the contents of the memory location in the operand are used.   
The memory address of the operand is specified indirectly through a register or a memory location. For example, in the instruction LOAD R1, [R2], the value in the memory location pointed to by the content of register R2 is loaded into register R1.

**Indexed addressing** – the contents of the memory location found by adding the contents of the index register (IR) to the address of the memory location in the operand are used..

The memory address of the operand is calculated by adding an offset to a base address. This addressing mode is useful for accessing elements of arrays or data structures. For example, in the instruction LOAD R1, [R2 + #8], the value at the memory location (R2 + 8) is loaded into register R1.

**Immediate addressing** – the value of the operand only is used.   
The operand is a constant value directly specified in the instruction. For example, MOV AX, 10 loads the value 10 directly into the AX register.

**Relative addressing** – the memory address used is the current memory address added to the operand.  
The memory address of the operand is calculated relative to the current program counter (PC) or instruction pointer. This addressing mode is typically used for control flow instructions. For example, in the instruction JUMP $+10, the program jumps to the instruction located 10 bytes ahead of the current instruction.

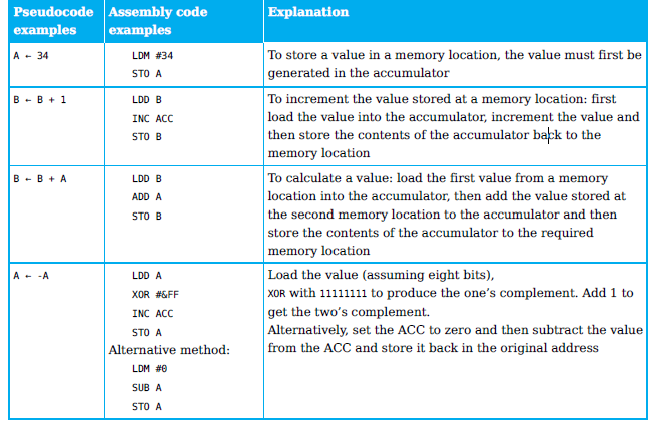
**Symbolic addressing** – only used in assembly language programming. A label is used instead of a value. allows programmers to use meaningful labels or symbols to represent memory addresses or jump targets instead of specifying the actual memory addresses directly.

Problem-solving and assembly-language programs

When writing a solution to a problem using low-level programming, we need to break down the problem into simple steps that can be programmed using the instruction set available.

Examples of assembly language assignments that match the pseudocode

Using the assignment



Exercise

Write assembly code instructions for this sequence of pseudocode statements:

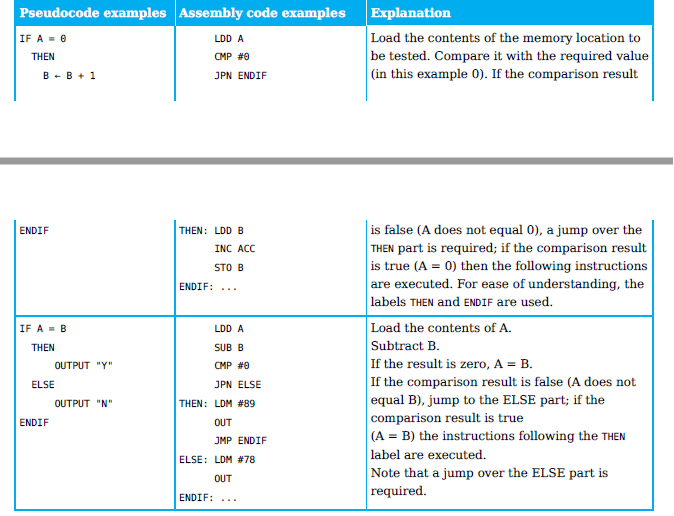
A ← 2

B ← 10

C ← A + B

D ← A - B

Using conditional statements



Exercise

Write assembly code instructions for this sequence of pseudocode statements:

IF A <> 0

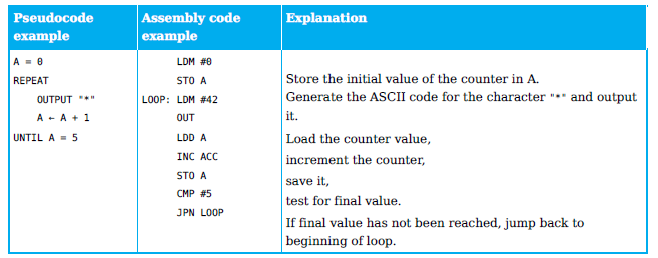
THEN

B ← A

ELSE

B ← B – 1

Example of repetition in assembly language that matches the pseudocode.



Exercise

Write assembly code instructions for this sequence of pseudocode statements:

Number ← 1

Total ← 0

Max ← 5

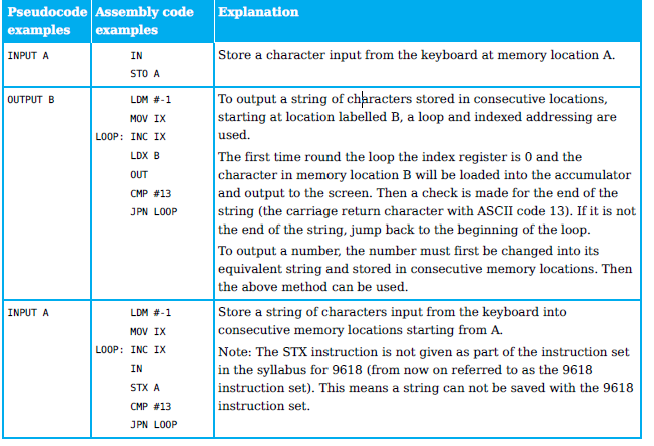
REPEAT

Total ← Total + Number

Number ← Number + 1

UNTIL Number = Max

examples of input and output in assembly language that match the pseudocode.



Write assembly code instructions for this sequence of pseudocode statements:

Count ← 0

REPEAT

Count ← Count + 1

INPUT Character

UNTIL Character = "N"

Modify the above assembly code instructions to implement this sequence of pseudocode statements:

Count ← 0

REPEAT

Count ← Count + 1

INPUT Character

UNTIL Character = "N"

OUTPUT Count