Machine language

Sometimes referred to as **machine code** or **object code**, **machine language** is a collection of binary digits or bits that the computer reads and interprets. Machine language is the only language a computer is capable of understanding.

Computer programs are written in one or more programming languages, like C++, Java, or Visual Basic. A computer cannot directly understand the programming languages used to create computer programs, so the program code must be compiled. Once a program's code is compiled, the computer can understand it because the program's code has been turned into machine language.

Assembly Language

Each personal computer has a microprocessor that manages the computer's arithmetical, logical, and control activities.

Each family of processors has its own set of instructions for handling various operations such as getting input from keyboard, displaying information on screen and performing various other jobs. These set of instructions are called 'machine language instructions'.

A processor understands only machine language instructions, which are strings of 1's and 0's. However, machine language is too obscure and complex for using in software development. So, the low-level assembly language is designed for a specific family of processors that represents various instructions in symbolic code and a more understandable form.

Advantages of Assembly Language

Having an understanding of assembly language makes one aware of :

- How programs interface with OS, processor, and BIOS;
- How data is represented in memory and other external devices;
- How the processor accesses and executes instruction;
- How instructions access and process data;
- How a program accesses external devices.

Other advantages of using assembly language are −

- It requires less memory and execution time;
- It allows hardware-specific complex jobs in an easier way;
- It is suitable for time-critical jobs;
- It is most suitable for writing interrupt service routines and other memory resident programs.

Basic Features of PC Hardware

The main internal hardware of a PC consists of processor, memory, and registers. Registers are processor components that hold data and address. To execute a program, the system copies it from the external device into the internal memory. The processor executes the program instructions.

The fundamental unit of computer storage is a bit; it could be ON (1) or OFF (0). A group of nine related bits makes a byte, out of which eight bits are used for data and the last one is used for parity. According to the rule of parity, the number of bits that are ON (1) in each byte should always be odd.

So, the parity bit is used to make the number of bits in a byte odd. If the parity is even, the system assumes that there had been a parity error (though rare), which might have been caused due to hardware fault or electrical disturbance.

The processor supports the following data sizes −

- Word: a 2-byte data item
- Doubleword: a 4-byte (32 bit) data item
- Quadword: an 8-byte (64 bit) data item
- Paragraph: a 16-byte (128 bit) area
- Kilobyte: 1024 bytes
- Megabyte: 1,048,576 bytes

Binary Number System

Every number system uses positional notation, i.e., each position in which a digit is written has a different positional value. Each position is power of the base, which is 2 for binary number system, and these powers begin at 0 and increase by 1.

The following table shows the positional values for an 8-bit binary number, where all bits are set ON.

Bit value	1	1	1	1	1	1	1	1
Position value as a power of base 2	128	64	32	16	8	4	2	1
Bit number	7	6	5	4	3	2	1	0

The value of a binary number is based on the presence of 1 bits and their positional value. So, the value of a given binary number is −

1 + 2 + 4 + 8 +16 + 32 + 64 + 128 = 255

which is same as 2^8 - 1.

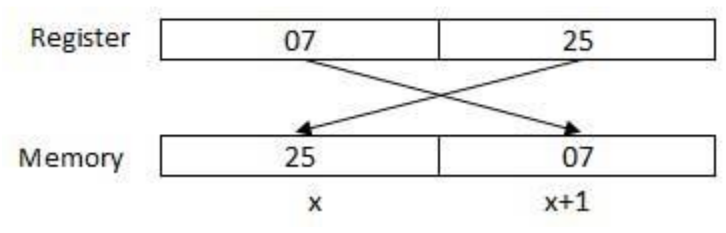
Addressing Data in Memory

The process through which the processor controls the execution of instructions is referred as the **fetch-decode-execute cycle** or the **execution cycle**. It consists of three continuous steps:

- Fetching the instruction from memory
- Decoding or identifying the instruction
- Executing the instruction

The processor may access one or more bytes of memory at a time. Let us consider a hexadecimal number 0725H. This number will require two bytes of memory. The high-order byte or most significant byte is 07 and the low-order byte is 25.

The processor stores data in reverse-byte sequence, i.e., a low-order byte is stored in a low memory address and a high-order byte in high memory address. So, if the processor brings the value 0725H from register to memory, it will transfer 25 first to the lower memory address and 07 to the next memory address.



x: memory address

When the processor gets the numeric data from memory to register, it again reverses the bytes. There are two kinds of memory addresses:

- Absolute address - a direct reference of specific location.
- Segment address (or offset) - starting address of a memory segment with the offset value.

Computer Registers

A register is a very small amount of very fast memory that is built into the CPU (central processing unit) in order to speed up its operations by providing quick access to commonly used values. Registers refers to semiconductor devices whose contents can be accessed (i.e., read and written to) at extremely high speeds but which are held there only temporarily (i.e., while in use or only as long as the power supply remains on).

Registers are the top of the memory hierarchy and are the fastest way for the system to manipulate data. Registers are normally measured by the number of bits they can hold, for example, an 8-bit register means it can store 8 bits of data or a 32-bit register means it can store 32 bit of data.

Registers are used to store data temporarily during the execution of a program. Some of the registers are accessible to the user through instructions. Data and instructions must be put into the system. So we need registers for this.

The basic computer registers with their names, size and functions are listed below

Register Symbol	**Register Name**	**Number of Bits**	**Description**
AC	Accumulator	16	Processor Register
DR	Data Register	16	Hold memory data
TR	Temporary Register	16	Holds temporary Data
IR	Instruction Register	16	Holds Instruction Code
AR	Address Register	12	Holds memory address
PC	Program Counter	12	Holds address of next instruction
INPR	Input Register	8	Holds Input data

OUTR	Output Register	8	Holds Output data

Programming language

Is a formal computer language designed to communicate instructions to a machine, particularly a computer. Programming languages can be used to create programs to control the behavior of a machine or to express algorithms.

Standard library and run-time system

Most programming languages have an associated core library (sometimes known as the 'standard library', especially if it is included as part of the published language standard), which is conventionally made available by all implementations of the language. Core libraries typically include definitions for commonly used algorithms, data structures, and mechanisms for input and output.

The line between a language and its core library differs from language to language. In some cases, the language designers may treat the library as a separate entity from the language. However, a language's core library is often treated as part of the language by its users, and some language specifications even require that this library be made available in all implementations.

Indeed, some languages are designed so that the meanings of certain syntactic constructs cannot even be described without referring to the core library. For example, in Java, a string literal is defined as an instance of the java.lang.String class; similarly, in Smalltalk, an anonymous function expression (a "block") constructs an instance of the library's BlockContext class. Conversely, Scheme contains multiple coherent subsets that suffice to construct the rest of the language as library macros, and so the language designers do not even bother to say which portions of the language must be implemented as language constructs, and which must be implemented as parts of a library.

Design and implementation

Programming languages share properties with natural languages related to their purpose as vehicles for communication, having a syntactic form separate from its semantics, and showing *language families* of related languages branching one from another. But as artificial constructs, they also differ in fundamental ways from languages that have evolved through usage. A significant difference is that a programming language can be fully described and studied in its entirety, since it has a precise and finite definition. By contrast, natural languages have changing meanings given by their users in different communities. While constructed languages are also artificial languages designed from the ground up with a specific purpose, they lack the precise and complete semantic definition that a programming language has.

Many programming languages have been designed from scratch, altered to meet new needs, and combined with other languages. Many have eventually fallen into disuse. Although there have been attempts to design one "universal" programming language that serves all purposes, all of them have failed to be generally accepted as filling this role. The need for diverse programming languages arises from the diversity of contexts in which languages are used:

- Programs range from tiny scripts written by individual hobbyists to huge systems written by hundreds of programmers.
- Programmers range in expertise from novices who need simplicity above all else, to experts who may be comfortable with considerable complexity.
- Programs must balance speed, size, and simplicity on systems ranging from microcontrollers to supercomputers.
- Programs may be written once and not change for generations, or they may undergo continual modification.
- Programmers may simply differ in their tastes: they may be accustomed to discussing problems and expressing them in a particular language.

One common trend in the development of programming languages has been to add more ability to solve problems using a higher level of abstraction. The earliest programming languages were tied very closely to the underlying hardware of the computer. As new programming languages have developed, features have been added that let programmers express ideas that are more remote from simple translation into underlying hardware instructions. Because programmers are less tied to the complexity of the computer, their programs can do more computing with less effort from the programmer. This lets them write more functionality per time unit.

Natural language programming has been proposed as a way to eliminate the need for a specialized language for programming. However, this goal remains distant and its benefits are open to debate. Edsger W. Dijkstra took the position that the use of a formal language is essential to prevent the introduction of meaningless constructs, and dismissed natural language programming as "foolish". Alan Perlis was similarly dismissive of the idea. Hybrid approaches have been taken in Structured English and SQL.

A language's designers and users must construct a number of artifacts that govern and enable the practice of programming. The most important of these artifacts are the language *specification* and *implementation*.

Specification

The specification of a programming language is an artifact that the language users and the implementors can use to agree upon whether a piece of source code is a valid program in that language, and if so what its behavior shall be.

A programming language specification can take several forms, including the following:

- An explicit definition of the syntax, static semantics, and execution semantics of the language. While syntax is commonly specified using a formal grammar, semantic definitions may be written in natural language (e.g., as in the C language), or a formal semantics (e.g., as in Standard ML and Scheme specifications).
- A description of the behavior of a translator for the language (e.g., the C++ and Fortran specifications). The syntax and semantics of the language have to be inferred from this description, which may be written in natural or a formal language.

- A *reference* or *model* implementation, sometimes written in the language being specified (e.g., Prolog or ANSI REXX). The syntax and semantics of the language are explicit in the behavior of the reference implementation.

Implementation

An implementation of a programming language provides a way to write programs in that language and execute them on one or more configurations of hardware and software. There are, broadly, two approaches to programming language implementation: compilation and interpretation. It is generally possible to implement a language using either technique.

The output of a compiler may be executed by hardware or a program called an interpreter. In some implementations that make use of the interpreter approach there is no distinct boundary between compiling and interpreting. For instance, some implementations of BASIC compile and then execute the source a line at a time.

Programs that are executed directly on the hardware usually run several orders of magnitude faster than those that are interpreted in software.

One technique for improving the performance of interpreted programs is just-in-time compilation. Here the virtual machine, just before execution, translates the blocks of bytecode which are going to be used to machine code, for direct execution on the hardware.

Proprietary languages

Although most of the most commonly used programming languages have fully open specifications and implementations, many programming languages exist only as proprietary programming languages with the implementation available only from a single vendor, which may claim that such a proprietary language is their intellectual property. Proprietary programming languages are commonly domain specific languages or internal scripting languages for a single product; some proprietary languages are used only internally within a vendor, while others are available to external users.

Some programming languages exist on the border between proprietary and open; for example, Oracle Corporation asserts proprietary rights to some aspects of the Java programming language, and Microsoft's C# programming language, which has open implementations of most parts of the system, also has Common Language Runtime (CLR) as a closed environment.

Many proprietary languages are widely used, in spite of their proprietary nature; examples include MATLAB and VBScript. Some languages may make the transition from closed to open; for example, Erlang was originally an Ericsson's internal programming language.

Usage

Thousands of different programming languages have been created, mainly in the computing field. Software is commonly built with 5 programming languages or more.

Programming languages differ from most other forms of human expression in that they require a greater degree of precision and completeness. When using a natural language to communicate with other people, human authors and speakers can be ambiguous and make small errors, and still expect their intent to be understood. However, figuratively speaking, computers "do exactly what they are told to do", and cannot "understand" what code the programmer intended to write. The combination of the language definition, a program, and the program's inputs must fully specify the external behavior that occurs when the program is executed, within the domain of control of that program. On the other hand, ideas about an algorithm can be communicated to humans without the precision required for execution by using pseudocode, which interleaves natural language with code written in a programming language.

A programming language provides a structured mechanism for defining pieces of data, and the operations or transformations that may be carried out automatically on that data. A programmer uses the abstractions present in the language to represent the concepts involved in a computation. These concepts are represented as a collection of the simplest elements available (called primitives). *Programming* is the process by which programmers combine these primitives to compose new programs, or adapt existing ones to new uses or a changing environment.

Programs for a computer might be executed in a batch process without human interaction, or a user might type commands in an interactive session of an interpreter. In this case the "commands" are simply programs, whose execution is chained together. When a language can run its commands through an interpreter (such as a Unix shell or other command-line interface), without compiling, it is called a scripting language

Measuring language usage

It is difficult to determine which programming languages are most widely used, and what usage means varies by context. One language may occupy the greater number of programmer hours, a different one have more lines of code, and a third may consume the most CPU time. Some languages are very popular for particular kinds of applications. For example, COBOL is still strong in the corporate data center, often on large mainframes; Fortran in scientific and engineering applications; Ada in aerospace, transportation, military, real-time and embedded applications; and C in embedded applications and operating systems. Other languages are regularly used to write many different kinds of applications.

Various methods of measuring language popularity, each subject to a different bias over what is measured, have been proposed:

- Counting the number of job advertisements that mention the language
- The number of books sold that teach or describe the language
- Estimates of the number of existing lines of code written in the language – which may underestimate languages not often found in public searches
- Counts of language references (i.e., to the name of the language) found using a web search engine.

Combining and averaging information from various internet sites, langpop.com claims that in 2013 the ten most popular programming languages are (in descending order by overall popularity): C, Java, PHP, JavaScript, C++, Python, Shell, Ruby, Objective-C and C#.

Virtual machine

In computing, a **virtual machine** (**VM**) is an emulation of a computer system. Virtual machines are based on computer architectures and provide functionality of a physical computer. Their implementations may involve specialized hardware, software, or a combination.

A VM or *virtual machine* was originally defined by Popek and Goldberg as "an efficient, isolated duplicate of a real computer machine." Current use includes virtual machines which have no direct correspondence to any real hardware.

There are different kinds of virtual machines, each with different functions:

- **System virtual machines** (also termed full virtualization VMs) provide a substitute for a real machine. They provide functionality needed to execute entire operating systems. A hypervisor uses native execution to share and manage hardware, allowing for multiple environments which are isolated from one another, yet exist on the same physical machine. Modern hypervisors use hardware-assisted virtualization, virtualization-specific hardware, primarily from the host CPUs.
- **Process virtual machines** are designed to execute computer programs in a platform-independent environment.

Some virtual machines, such as QEMU, are designed to also emulate different architectures and allow execution of software applications and operating systems written for another CPU or architecture. Operating-system-level virtualization allows the resources of a computer to be partitioned via the kernel's support for multiple isolated user space instances, which are usually called containers and may look and feel like real machines to the end users.

System virtual machines

The desire to run multiple operating systems was the initial motive for virtual machines, so as to allow time-sharing among several single-tasking operating systems. In some respects, a system virtual machine can be considered a generalization of the concept of virtual memory that historically preceded it. IBM's CP/CMS, the first systems to allow full virtualization, implemented time sharing by providing each user with a single-user operating system, the Conversational Monitor System (CMS). Unlike virtual memory, a system virtual machine entitled the user to write privileged instructions in their code. This approach had certain advantages, such as adding input/output devices not allowed by the standard system.

As technology evolves virtual memory for purposes of virtualization, new systems of memory over commitment may be applied to manage memory sharing among multiple virtual machines on one computer operating system. It may be possible to share *memory pages* that have identical contents among multiple virtual machines that run on the same physical machine, what may result in mapping them to the same physical page by a technique termed Kernel Same Page Merging. This is especially useful for read-only pages, such as those holding code segments, which is the case for multiple virtual machines running the same or similar software, software libraries, web servers, middleware components, etc. The guest operating systems do not need to be compliant with the host hardware, thus making it possible to run different operating systems on the same computer (e.g., Windows, Linux, or prior versions of an operating system) to support future software.

The use of virtual machines to support separate guest operating systems is popular in regard to embedded systems. A typical use would be to run a real-time operating system simultaneously with a preferred complex operating system, such as Linux or Windows. Another use would be for novel and unproven software still in the developmental stage, so it runs inside a sandbox. Virtual machines have other advantages for operating system development, and may include improved debugging access and faster reboots.

Multiple VMs running their own guest operating system are frequently engaged for server consolidation.

Full virtualization

In full virtualization, the virtual machine simulates enough hardware to allow an unmodified "guest" OS (one designed for the same instruction set) to be run in isolation. This approach was pioneered in 1966 with the IBM CP-40 and CP-67, predecessors of the VM family.

Examples outside the mainframe field include Parallels Workstation, Parallels Desktop for Mac, VirtualBox, Virtual Iron, Oracle VM, Virtual PC, Virtual Server, Hyper-V, VMware Workstation, VMware Server (discontinued, formerly called GSX Server), VMware ESXi, QEMU, Adeos, Mac-on-Linux, Win4BSD, Win4Lin Pro, and Egenera vBlade technology.

Hardware-assisted virtualization

In hardware-assisted virtualization, the hardware provides architectural support that facilitates building a virtual machine monitor and allows guest OSes to be run in isolation. Hardware-assisted virtualization was first introduced on the IBM System/370 in 1972, for use with VM/370, the first virtual machine operating system.

In 2005 and 2006, Intel and AMD provided additional hardware to support virtualization. Sun Microsystems (now Oracle Corporation) added similar features in their UltraSPARC T-Series processors in 2005. Examples of virtualization platforms adapted to such hardware include KVM, VMware Workstation, VMware Fusion, Hyper-V, Windows Virtual PC, Xen, Parallels Desktop for Mac, Oracle VM Server for SPARC, VirtualBox and Parallels Workstation.

In 2006, first-generation 32- and 64-bit x86 hardware support was found to rarely offer performance advantages over software virtualization

Operating-system-level virtualization

In operating-system-level virtualization, a physical server is virtualized at the operating system level, enabling multiple isolated and secure virtualized servers to run on a single physical server. The "guest" operating system environments share the same running instance of the operating system as the host system. Thus, the same operating system kernel is also used to implement the "guest" environments, and applications running in a given "guest" environment view it as a stand-alone system. The pioneer implementation was FreeBSD jails; other examples include Docker, Solaris Containers, OpenVZ, Linux-VServer, LXC, AIX Workload Partitions, Parallels Virtuozzo Containers, and iCore Virtual Accounts.