Assembler language, often abbreviated asm, is any low-level programming language in which there is a very strong correspondence between the instructions in the language and the architecture's machine code instructions. Because assembly depends on the machine code instructions, every assembly language is designed for exactly one specific computer architecture. Assembly language may also be called symbolic machine code.

Assembly code is converted into executable machine code by a utility program referred to as an assembler. The conversion process is referred to as assembly, as in assembling the source code. Assembly language usually has one statement per machine instruction (1:1), but comments and statements that are assembler directives, macros, and symbolic labels of program and memory locations are often also supported.

An assembler program creates object code by translating combinations of mnemonics and syntax for operations and addressing modes into their numerical equivalents. This representation typically includes an operation code ("opcode") as well as other control bits and data. The assembler also calculates constant expressions and resolves symbolic names for memory locations and other entities. The use of symbolic references is a key feature of assemblers, saving tedious calculations and manual address updates after program modifications. Most assemblers also include macro facilities for performing textual substitution – e.g., to generate common short sequences of instructions as inline, instead of called subroutines.

Some assemblers may also be able to perform some simple types of instruction set-specific optimizations. One concrete example of this may be the ubiquitous x86 assemblers from various vendors. Called jump-sizing, most of them are able to perform jump-instruction replacements (long jumps replaced by short or relative jumps) in any number of passes, on request. Others may even do simple rearrangement or insertion of instructions, such as some assemblers for RISC architectures that can help optimize a sensible instruction scheduling to exploit the CPU pipeline as efficiently as possible.

There are two types of assemblers based on how many passes through the source are needed (how many times the assembler reads the source) to produce the object file.

One-pass assemblers go through the source code once. Any symbol used before it is defined will require "errata" at the end of the object code (or, at least, no earlier than the point where the symbol is defined) telling the linker or the loader to "go back" and overwrite a placeholder which had been left where the as yet undefined symbol was used.

Multi-pass assemblers create a table with all symbols and their values in the first passes, then use the table in later passes to generate code.

In both cases, the assembler must be able to determine the size of each instruction on the initial passes in order to calculate the addresses of subsequent symbols. This means that if the size of an operation referring to an operand defined later depends on the type or distance of the operand, the assembler will make a pessimistic estimate when first encountering the operation, and if necessary, pad it with one or more "no-operation" instructions in a later pass or the errata. In an assembler with peephole optimization, addresses may be recalculated between passes to allow replacing pessimistic code with code tailored to the exact distance from the target.

The original reason for the use of one-pass assemblers was memory size and speed of assembly – often a second pass would require storing the symbol table in memory (to handle forward references), rewinding and rereading the program source on tape, or rereading a deck of cards or punched paper tape. Later computers with much larger memories (especially disc storage), had the space to perform all necessary processing without such re-reading. The advantage of the multi-pass assembler is that the absence of errata makes the linking process (or the program load if the assembler directly produces executable code) faster.

Assembly directives, also called pseudo-opcodes, pseudo-operations or pseudo-ops, are commands given to an assembler "directing it to perform operations other than assembling instructions". Directives affect how the assembler operates and "may affect the object code, the symbol table, the listing file, and the values of internal assembler parameters". Sometimes the term pseudo-opcode is reserved for directives that generate object code, such as those that generate data.

Many assemblers support predefined macros, and others support programmer-defined (and repeatedly re-definable) macros involving sequences of text lines in which variables and constants are embedded. The macro definition is most commonly a mixture of assembler statements, e.g., directives, symbolic machine instructions, and templates for assembler statements. This sequence of text lines may include opcodes or directives. Once a macro has been defined its name may be used in place of a mnemonic. When the assembler processes such a statement, it replaces the statement with the text lines associated with that macro, then processes them as if they existed in the source code file (including, in some assemblers, expansion of any macros existing in the replacement text).

Assembly language is typically used in a system's boot code, the low-level code that initializes and tests the system hardware prior to booting the operating system and is often stored in ROM. (BIOS on IBM-compatible PC systems and CP/M is an example.)Some compilers for relatively low-level languages, such as Pascal or C, allow the programmer to embed assembly language directly in the source code (so called inline assembly). Programs using such facilities can then construct abstractions using different assembly language on each hardware platform. The system's portable code can then use these processor-specific components through a uniform interface. Assembly language is also useful in reverse engineering. Many programs are distributed only in machine code form which is straightforward to translate into assembly language by a disassembler, but more difficult to translate into a higher-level language through a decompiler. The disassembly technique is used by hackers to crack commercial software, and competitors to produce software with similar results from competing companies.