Assembly language

From Wikipedia, the free encyclopedia

[Jump to navigation](https://en.wikipedia.org/wiki/Assembly_language#mw-head)[Jump to search](https://en.wikipedia.org/wiki/Assembly_language#p-search)

An **assembly** (or **assembler**) **language**,[[1]](https://en.wikipedia.org/wiki/Assembly_language#cite_note-ASM1-1) often abbreviated **asm**, is any [low-level programming language](https://en.wikipedia.org/wiki/Low-level_programming_language) in which there is a very strong correspondence between the program's statements and the [architecture's](https://en.wikipedia.org/wiki/Computer_architecture) [machine code](https://en.wikipedia.org/wiki/Machine_code) [instructions](https://en.wikipedia.org/wiki/Instruction_set_architecture).[[2]](https://en.wikipedia.org/wiki/Assembly_language#cite_note-2)

Each assembly language is specific to a particular computer architecture and operating system.[[3]](https://en.wikipedia.org/wiki/Assembly_language#cite_note-3) In contrast, most [high-level programming languages](https://en.wikipedia.org/wiki/High-level_programming_language) are generally [portable](https://en.wikipedia.org/wiki/Porting) across multiple architectures but require [interpreting](https://en.wikipedia.org/wiki/Interpreter_(computing)) or [compiling](https://en.wikipedia.org/wiki/Compiler). Assembly language may also be called *symbolic machine code*.[[4]](https://en.wikipedia.org/wiki/Assembly_language#cite_note-4)[[5]](https://en.wikipedia.org/wiki/Assembly_language#cite_note-5)

Assembly language usually has one statement per machine instruction, but assembler directives,[[6]](https://en.wikipedia.org/wiki/Assembly_language#cite_note-A.DIR-6) [macros](https://en.wikipedia.org/wiki/Macro_instruction)[[7]](https://en.wikipedia.org/wiki/Assembly_language#cite_note-7)[[1]](https://en.wikipedia.org/wiki/Assembly_language#cite_note-ASM1-1) and symbolic labels of program and memory locations are often also supported.

Assembly code is converted into executable machine code by a [utility program](https://en.wikipedia.org/wiki/Utility_software) referred to as an [*assembler*](https://en.wikipedia.org/wiki/Assembly_language#Assembler). The conversion process is referred to as *assembly*, or *assembling* the [source code](https://en.wikipedia.org/wiki/Source_code).

Assembler language syntax



Assembly language uses a [mnemonic](https://en.wikipedia.org/wiki/Mnemonic) to represent each low-level [machine instruction](https://en.wikipedia.org/wiki/Machine_code) or [opcode](https://en.wikipedia.org/wiki/Opcode), typically also each [architectural register](https://en.wikipedia.org/wiki/Register_(computing)#ARCHITECTURAL), [flag](https://en.wikipedia.org/wiki/Bit_field), etc. Many operations require one or more [operands](https://en.wikipedia.org/wiki/Operand#Computer_science) in order to form a complete instruction. Most assemblers permit named constants, registers, and [labels](https://en.wikipedia.org/wiki/Label_(computer_science)) for program and memory locations, and can calculate [expressions](https://en.wikipedia.org/wiki/Expression_(computer_science)) for operands. Thus, the programmers are freed from tedious repetitive calculations and assembler programs are much more readable than machine code. Depending on the architecture, these elements may also be combined for specific instructions or [addressing modes](https://en.wikipedia.org/wiki/Addressing_mode) using [offsets](https://en.wikipedia.org/wiki/Offset_(computer_science)) or other data as well as fixed addresses. Many assemblers offer additional mechanisms to facilitate program development, to control the assembly process, and to aid [debugging](https://en.wikipedia.org/wiki/Debugging).

Terminology

* A **macro assembler** includes a [macroinstruction](https://en.wikipedia.org/wiki/Macro_(computer_science)) facility so that (parameterized) assembly language text can be represented by a name, and that name can be used to insert the expanded text into other code.
* A **cross assembler** (see also [cross compiler](https://en.wikipedia.org/wiki/Cross_compiler)) is an assembler that is run on a computer or [operating system](https://en.wikipedia.org/wiki/Operating_system) (the *host* system) of a different type from the system on which the resulting code is to run (the *target system*). Cross-assembling facilitates the development of programs for systems that do not have the resources to support software development, such as an [embedded system](https://en.wikipedia.org/wiki/Embedded_system). In such a case, the resulting [object code](https://en.wikipedia.org/wiki/Object_code) must be transferred to the target system, either via [read-only memory](https://en.wikipedia.org/wiki/Read-only_memory) (ROM, [EPROM](https://en.wikipedia.org/wiki/EPROM), etc.) or a data link using an exact bit-by-bit copy of the object code or a text-based representation of that code, such as [Motorola S-record](https://en.wikipedia.org/wiki/SREC_(file_format)) or [Intel HEX](https://en.wikipedia.org/wiki/Intel_HEX).
* A [**high-level assembler**](https://en.wikipedia.org/wiki/High-level_assembler) is a program that provides language abstractions more often associated with high-level languages, such as advanced control structures ([IF/THEN/ELSE](https://en.wikipedia.org/wiki/If-then-else), DO CASE, etc.) and high-level abstract data types, including structures/records, unions, classes, and sets.
* A **[microassembler](https://en.wikipedia.org/wiki/Microassembler" \o "Microassembler)** is a program that helps prepare a [microprogram](https://en.wikipedia.org/wiki/Microcode), called *firmware*, to control the low level operation of a computer.
* A **meta-assembler** is a term used in some circles for *"a program that accepts the syntactic and semantic description of an assembly language, and generates an assembler for that language."*[[8]](https://en.wikipedia.org/wiki/Assembly_language#cite_note-8)
* *Assembly time* is the computational step where an assembler is run.

Key concepts

**Assembler**

An **assembler** program creates [object code](https://en.wikipedia.org/wiki/Object_code) by translating combinations of mnemonics and [syntax](https://en.wikipedia.org/wiki/Syntax) for operations and addressing modes into their numerical equivalents. This representation typically includes an *operation code* ("[opcode](https://en.wikipedia.org/wiki/Opcode)") as well as other control [bits](https://en.wikipedia.org/wiki/Bit) and data. The assembler also calculates constant expressions and resolves [symbolic names](https://en.wikipedia.org/wiki/Identifier) for memory locations and other entities.[[9]](https://en.wikipedia.org/wiki/Assembly_language#cite_note-Salomon-9) 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](https://en.wikipedia.org/wiki/Macro_(computer_science)) facilities for performing textual substitution – e.g., to generate common short sequences of instructions as [inline](https://en.wikipedia.org/wiki/Inline_expansion), instead of *called* [subroutines](https://en.wikipedia.org/wiki/Subroutine).

Some assemblers may also be able to perform some simple types of [instruction set](https://en.wikipedia.org/wiki/Instruction_set)-specific [optimizations](https://en.wikipedia.org/wiki/Compiler_optimization). One concrete example of this may be the ubiquitous [x86](https://en.wikipedia.org/wiki/X86) assemblers from various vendors. 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](https://en.wikipedia.org/wiki/RISC) [architectures](https://en.wikipedia.org/wiki/Instruction_set_architecture) that can help optimize a sensible [instruction scheduling](https://en.wikipedia.org/wiki/Instruction_scheduling) to exploit the [CPU pipeline](https://en.wikipedia.org/wiki/CPU_pipeline) as efficiently as possible.[[*citation needed*](https://en.wikipedia.org/wiki/Wikipedia:Citation_needed)]

Like early programming languages such as [Fortran](https://en.wikipedia.org/wiki/Fortran), [Algol](https://en.wikipedia.org/wiki/ALGOL), [Cobol](https://en.wikipedia.org/wiki/Cobol) and [Lisp](https://en.wikipedia.org/wiki/Lisp_(programming_language)), assemblers have been available since the 1950s and the first generations of text based [computer interfaces](https://en.wikipedia.org/wiki/Computer_interface). However, assemblers came first as they are far simpler to write than [compilers](https://en.wikipedia.org/wiki/Compiler) for [high-level languages](https://en.wikipedia.org/wiki/High-level_language). This is because each mnemonic along with the addressing modes and operands of an instruction translates rather directly into the numeric representations of that particular instruction, without much context or analysis. There have also been several classes of translators and semi automatic [code generators](https://en.wikipedia.org/wiki/Code_generator) with properties similar to both assembly and [high level languages](https://en.wikipedia.org/wiki/High_level_language), with [speedcode](https://en.wikipedia.org/wiki/Speedcode" \o "Speedcode) as perhaps one of the better known examples.

There may be several assemblers with different [syntax](https://en.wikipedia.org/wiki/Syntax_(programming_languages)) for a particular [CPU](https://en.wikipedia.org/wiki/Central_processing_unit) or [instruction set architecture](https://en.wikipedia.org/wiki/Instruction_set_architecture). For instance, an instruction to add memory data to a register in a [x86](https://en.wikipedia.org/wiki/X86)-family processor might be add eax,[ebx], in original [*Intel syntax*](https://en.wikipedia.org/wiki/Intel_syntax), whereas this would be written addl (%ebx),%eax in the [*AT&T syntax*](https://en.wikipedia.org/wiki/AT%26T_syntax) used by the [GNU Assembler](https://en.wikipedia.org/wiki/GNU_Assembler). Despite different appearances, different syntactic forms generally generate the same numeric [machine code](https://en.wikipedia.org/wiki/Machine_code), see further below. A single assembler may also have different modes in order to support variations in syntactic forms as well as their exact semantic interpretations (such as [FASM](https://en.wikipedia.org/wiki/FASM)-syntax, [TASM](https://en.wikipedia.org/wiki/TASM)-syntax, ideal mode etc., in the special case of [x86 assembly](https://en.wikipedia.org/wiki/X86_assembly_language)programming).

**Number of passes**

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"](https://en.wikipedia.org/wiki/Erratum) at the end of the object code (or, at least, no earlier than the point where the symbol is defined) telling the [linker](https://en.wikipedia.org/wiki/Linker_(computing)) 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](https://en.wikipedia.org/wiki/NOP_(code))" instructions in a later pass or the errata. In an assembler with [peephole optimization](https://en.wikipedia.org/wiki/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 speed of assembly – often a second pass would require rewinding and rereading the program source on [tape](https://en.wikipedia.org/wiki/Magnetic_tape_data_storage) or rereading a deck of [cards](https://en.wikipedia.org/wiki/Punch_cards) or [punched paper tape](https://en.wikipedia.org/wiki/Punched_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](https://en.wikipedia.org/wiki/Linker_(computing)) (or the [program load](https://en.wikipedia.org/wiki/Loader_(computing)) if the assembler directly produces executable code) faster.[[10]](https://en.wikipedia.org/wiki/Assembly_language#cite_note-10)

**Example:** in the following code snippet a one-pass assembler would be able to determine the address of the backward reference *BKWD* when assembling statement *S2*, but would not be able to determine the address of the forward reference *FWD* when assembling the branch statement *S1*; indeed *FWD* may be undefined. A two-pass assembler would determine both addresses in pass 1, so they would be known when generating code in pass 2,

**High-level assemblers**

More sophisticated [high-level assemblers](https://en.wikipedia.org/wiki/High-level_assembler) provide language abstractions such as:

* High-level procedure/function declarations and invocations
* Advanced control structures (IF/THEN/ELSE, SWITCH)
* High-level abstract data types, including structures/records, unions, classes, and sets
* Sophisticated macro processing (although available on ordinary assemblers since the late 1950s for [IBM 700 series](https://en.wikipedia.org/wiki/IBM_700/7000_series) and since the 1960s for [IBM/360](https://en.wikipedia.org/wiki/IBM/360), amongst other machines)
* [Object-oriented programming](https://en.wikipedia.org/wiki/Object-oriented_programming) features such as [classes](https://en.wikipedia.org/wiki/Class_(computer_programming)), [objects](https://en.wikipedia.org/wiki/Object_(computer_science)), [abstraction](https://en.wikipedia.org/wiki/Abstraction_(computer_science)), [polymorphism](https://en.wikipedia.org/wiki/Type_polymorphism), and [inheritance](https://en.wikipedia.org/wiki/Inheritance_(object-oriented_programming))[[11]](https://en.wikipedia.org/wiki/Assembly_language#cite_note-11)

See [Language design](https://en.wikipedia.org/wiki/Assembly_language#Language_design) below for more details.

**Assembly language**

A program written in assembly language consists of a series of [mnemonic](https://en.wikipedia.org/wiki/Mnemonic) processor instructions and meta-statements (known variously as directives, pseudo-instructions and pseudo-ops), comments and data. Assembly language instructions usually consist of an [opcode](https://en.wikipedia.org/wiki/Opcode) mnemonic followed by a list of data, arguments or parameters.[[12]](https://en.wikipedia.org/wiki/Assembly_language#cite_note-intel-1999-12) These are translated by an [assembler](https://en.wikipedia.org/wiki/Assembly_language_assembler) into [machine language](https://en.wikipedia.org/wiki/Machine_language) instructions that can be loaded into memory and executed.

For example, the instruction below tells an [x86](https://en.wikipedia.org/wiki/X86)/[IA-32](https://en.wikipedia.org/wiki/IA-32) processor to move an [immediate 8-bit value](https://en.wikipedia.org/wiki/Constant_(programming)) into a [register](https://en.wikipedia.org/wiki/Processor_register). The binary code for this instruction is 10110 followed by a 3-bit identifier for which register to use. The identifier for the *AL* register is 000, so the following [machine code](https://en.wikipedia.org/wiki/Machine_code) loads the *AL* register with the data 01100001.[[13]](https://en.wikipedia.org/wiki/Assembly_language#cite_note-intel-1999-MOV-13)

Here, B0 means 'Move a copy of the following value into *AL'*, and 61 is a hexadecimal representation of the value 01100001, which is 97 in [decimal](https://en.wikipedia.org/wiki/Decimal). Assembly language for the 8086 family provides the [mnemonic](https://en.wikipedia.org/wiki/Mnemonic) [MOV](https://en.wikipedia.org/wiki/MOV_(x86_instruction)) (an abbreviation of *move*) for instructions such as this, so the machine code above can be written as follows in assembly language, complete with an explanatory comment if required, after the semicolon. This is much easier to read and to remember.

In some assembly languages the same mnemonic such as MOV may be used for a family of related instructions for loading, copying and moving data, whether these are immediate values, values in registers, or memory locations pointed to by values in registers. Other assemblers may use separate opcode mnemonics such as L for "move memory to register", ST for "move register to memory", LR for "move register to register", MVI for "move immediate operand to memory", etc.

The x86 opcode 10110000 (B0) copies an 8-bit value into the *AL* register, while 10110001 (B1) moves it into *CL* and 10110010 (B2) does so into *DL*. Assembly language examples for these follow.[[13]](https://en.wikipedia.org/wiki/Assembly_language#cite_note-intel-1999-MOV-13)

The syntax of MOV can also be more complex as the following examples show.[[14]](https://en.wikipedia.org/wiki/Assembly_language#cite_note-14)

In each case, the MOV mnemonic is translated directly into an opcode in the ranges 88-8E, A0-A3, B0-B8, C6 or C7 by an assembler, and the programmer does not have to know or remember which.[[13]](https://en.wikipedia.org/wiki/Assembly_language#cite_note-intel-1999-MOV-13)

Transforming assembly language into machine code is the job of an assembler, and the reverse can at least partially be achieved by a [disassembler](https://en.wikipedia.org/wiki/Disassembler). Unlike [high-level languages](https://en.wikipedia.org/wiki/High-level_language), there is a [one-to-one correspondence](https://en.wikipedia.org/wiki/One-to-one_correspondence) between many simple assembly statements and machine language instructions. However, in some cases, an assembler may provide *pseudoinstructions* (essentially macros) which expand into several machine language instructions to provide commonly needed functionality. For example, for a machine that lacks a "branch if greater or equal" instruction, an assembler may provide a pseudoinstruction that expands to the machine's "set if less than" and "branch if zero (on the result of the set instruction)". Most full-featured assemblers also provide a rich [macro](https://en.wikipedia.org/wiki/Macro_(computer_science)) language (discussed below) which is used by vendors and programmers to generate more complex code and data sequences.

Each [computer architecture](https://en.wikipedia.org/wiki/Computer_architecture) has its own machine language. Computers differ in the number and type of operations they support, in the different sizes and numbers of registers, and in the representations of data in storage. While most general-purpose computers are able to carry out essentially the same functionality, the ways they do so differ; the corresponding assembly languages reflect these differences.

Multiple sets of [mnemonics](https://en.wikipedia.org/wiki/Mnemonic) or assembly-language syntax may exist for a single instruction set, typically instantiated in different assembler programs. In these cases, the most popular one is usually that supplied by the manufacturer and used in its documentation.

Language design

**Basic elements**

There is a large degree of diversity in the way the authors of assemblers categorize statements and in the nomenclature that they use. In particular, some describe anything other than a machine mnemonic or extended mnemonic as a pseudo-operation (pseudo-op). A typical assembly language consists of 3 types of instruction statements that are used to define program operations:

* [Opcode](https://en.wikipedia.org/wiki/Opcode) mnemonics
* Data definitions
* Assembly directives

**Opcode mnemonics and extended mnemonics**

Instructions (statements) in assembly language are generally very simple, unlike those in [high-level languages](https://en.wikipedia.org/wiki/High-level_programming_language). Generally, a mnemonic is a symbolic name for a single executable machine language instruction (an [opcode](https://en.wikipedia.org/wiki/Opcode)), and there is at least one opcode mnemonic defined for each machine language instruction. Each instruction typically consists of an *operation* or *opcode* plus zero or more [*operands*](https://en.wikipedia.org/wiki/Operand). Most instructions refer to a single value, or a pair of values. Operands can be immediate (value coded in the instruction itself), registers specified in the instruction or implied, or the addresses of data located elsewhere in storage. This is determined by the underlying processor architecture: the assembler merely reflects how this architecture works. *Extended mnemonics* are often used to specify a combination of an opcode with a specific operand, e.g., the System/360 assemblers use B as an extended mnemonic for BC with a mask of 15 and NOP ("NO OPeration" – do nothing for one step) for BC with a mask of 0.

*Extended mnemonics* are often used to support specialized uses of instructions, often for purposes not obvious from the instruction name. For example, many CPU's do not have an explicit NOP instruction, but do have instructions that can be used for the purpose. In 8086 CPUs the instruction xchg ax,ax is used for nop, with nop being a pseudo-opcode to encode the instruction xchg ax,ax. Some disassemblers recognize this and will decode the xchg ax,ax instruction as nop. Similarly, IBM assemblers for [System/360](https://en.wikipedia.org/wiki/IBM_System/360) and [System/370](https://en.wikipedia.org/wiki/IBM_System/370) use the extended mnemonics NOP and NOPR for BC and BCR with zero masks. For the SPARC architecture, these are known as *synthetic instructions*.[[15]](https://en.wikipedia.org/wiki/Assembly_language#cite_note-15)

Some assemblers also support simple built-in macro-instructions that generate two or more machine instructions. For instance, with some Z80 assemblers the instruction ld hl,bcis recognized to generate ld l,c followed by ld h,b.[[16]](https://en.wikipedia.org/wiki/Assembly_language#cite_note-16) These are sometimes known as *pseudo-opcodes*.

Mnemonics are arbitrary symbols; in 1985 the [IEEE](https://en.wikipedia.org/wiki/IEEE) published Standard 694 for a uniform set of mnemonics to be used by all assemblers. The standard has since been withdrawn.

**Data directives**

There are instructions used to define data elements to hold data and variables. They define the type of data, the length and the [alignment](https://en.wikipedia.org/wiki/Data_structure_alignment) of data. These instructions can also define whether the data is available to outside programs (programs assembled separately) or only to the program in which the data section is defined. Some assemblers classify these as pseudo-ops.

**Assembly directives**

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.".[[9]](https://en.wikipedia.org/wiki/Assembly_language#cite_note-Salomon-9) 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.[[17]](https://en.wikipedia.org/wiki/Assembly_language#cite_note-17)

The names of pseudo-ops often start with a dot to distinguish them from machine instructions. Pseudo-ops can make the assembly of the program dependent on parameters input by a programmer, so that one program can be assembled different ways, perhaps for different applications. Or, a pseudo-op can be used to manipulate presentation of a program to make it easier to read and maintain. Another common use of pseudo-ops is to reserve storage areas for run-time data and optionally initialize their contents to known values.

Symbolic assemblers let programmers associate arbitrary names ([*labels*](https://en.wikipedia.org/wiki/Label_(programming_language)) or *symbols*) with memory locations and various constants. Usually, every constant and variable is given a name so instructions can reference those locations by name, thus promoting [self-documenting code](https://en.wikipedia.org/wiki/Self-documenting_code). In executable code, the name of each subroutine is associated with its entry point, so any calls to a subroutine can use its name. Inside subroutines, [GOTO](https://en.wikipedia.org/wiki/GOTO) destinations are given labels. Some assemblers support *local symbols* which are lexically distinct from normal symbols (e.g., the use of "10$" as a GOTO destination).

Some assemblers, such as NASM, provide flexible symbol management, letting programmers manage different [namespaces](https://en.wikipedia.org/wiki/Namespace), automatically calculate offsets within [data structures](https://en.wikipedia.org/wiki/Data_structure), and assign labels that refer to literal values or the result of simple computations performed by the assembler. Labels can also be used to initialize constants and variables with relocatable addresses.

Assembly languages, like most other computer languages, allow comments to be added to program [source code](https://en.wikipedia.org/wiki/Source_code) that will be ignored during assembly. Judicious commenting is essential in assembly language programs, as the meaning and purpose of a sequence of binary machine instructions can be difficult to determine. The "raw" (uncommented) assembly language generated by compilers or disassemblers is quite difficult to read when changes must be made.

.[[21]](https://en.wikipedia.org/wiki/Assembly_language#cite_note-22)

**Support for structured programming**

Some assemblers have incorporated [structured programming](https://en.wikipedia.org/wiki/Structured_programming) elements to encode execution flow. The earliest example of this approach was in the [Concept-14 macro set](https://en.wikipedia.org/w/index.php?title=Concept-14_macro_set&action=edit&redlink=1), originally proposed by Dr. [Harlan Mills](https://en.wikipedia.org/wiki/Harlan_Mills) (March 1970), and implemented by Marvin Kessler at IBM's Federal Systems Division, which extended the S/360 macro assembler with IF/ELSE/ENDIF and similar control flow blocks.[[22]](https://en.wikipedia.org/wiki/Assembly_language#cite_note-23) This was a way to reduce or eliminate the use of [GOTO](https://en.wikipedia.org/wiki/GOTO) operations in assembly code, one of the main factors causing [spaghetti code](https://en.wikipedia.org/wiki/Spaghetti_code) in assembly language. This approach was widely accepted in the early '80s (the latter days of large-scale assembly language use).

A curious design was [A-natural](https://en.wikipedia.org/w/index.php?title=A-natural&action=edit&redlink=1), a "stream-oriented" assembler for 8080/[Z80](https://en.wikipedia.org/wiki/Z80) processors[[*citation needed*](https://en.wikipedia.org/wiki/Wikipedia:Citation_needed)] from [Whitesmiths Ltd.](https://en.wikipedia.org/wiki/Whitesmiths) (developers of the [Unix](https://en.wikipedia.org/wiki/Unix)-like [Idris](https://en.wikipedia.org/wiki/Idris_(operating_system)) operating system, and what was reported to be the first commercial [C](https://en.wikipedia.org/wiki/C_(programming_language)) [compiler](https://en.wikipedia.org/wiki/Compiler)). The language was classified as an assembler, because it worked with raw machine elements such as [opcodes](https://en.wikipedia.org/wiki/Opcodes), [registers](https://en.wikipedia.org/wiki/Processor_register), and memory references; but it incorporated an expression syntax to indicate execution order. Parentheses and other special symbols, along with block-oriented structured programming constructs, controlled the sequence of the generated instructions. A-natural was built as the object language of a C compiler, rather than for hand-coding, but its logical syntax won some fans.

There has been little apparent demand for more sophisticated assemblers since the decline of large-scale assembly language development.[[23]](https://en.wikipedia.org/wiki/Assembly_language#cite_note-assembly-language?cat=technology-24) In spite of that, they are still being developed and applied in cases where resource constraints or peculiarities in the target system's architecture prevent the effective use of higher-level languages.[[24]](https://en.wikipedia.org/wiki/Assembly_language#cite_note-25)

Use of assembly language

**Historical perspective**

Assembly languages date to the introduction of the [stored-program computer](https://en.wikipedia.org/wiki/Stored-program_computer). The first assembly language was developed in 1947 by [Kathleen Booth](https://en.wikipedia.org/wiki/Kathleen_Booth) for the [ARC2](https://en.wikipedia.org/wiki/APEXC) at [Birkbeck, University of London](https://en.wikipedia.org/wiki/Birkbeck,_University_of_London" \o "Birkbeck, University of London) following work with [John von Neumann](https://en.wikipedia.org/wiki/John_von_Neumann) and [Herman Goldstine](https://en.wikipedia.org/wiki/Herman_Goldstine) at the [Institute for Advanced Study](https://en.wikipedia.org/wiki/Institute_for_Advanced_Study).[[25]](https://en.wikipedia.org/wiki/Assembly_language#cite_note-26)[[26]](https://en.wikipedia.org/wiki/Assembly_language#cite_note-birkbeck-27) The [Electronic Delay Storage Automatic Calculator](https://en.wikipedia.org/wiki/Electronic_Delay_Storage_Automatic_Calculator)(EDSAC) had an assembler (named "initial orders") integrated into its [bootstrap](https://en.wikipedia.org/wiki/Booting) program that used one-letter mnemonics in late 1948 developed by [David Wheeler](https://en.wikipedia.org/wiki/David_Wheeler_(computer_scientist)) and credited by the IEEE Computer Society as the creator of the first "assembler."[[27]](https://en.wikipedia.org/wiki/Assembly_language#cite_note-28)[[28]](https://en.wikipedia.org/wiki/Assembly_language#cite_note-29) [[29]](https://en.wikipedia.org/wiki/Assembly_language#cite_note-30) Reports on the EDSAC introduced the term "assembly" for the process of combining fields into an instruction word.[[30]](https://en.wikipedia.org/wiki/Assembly_language#cite_note-31) SOAP (Symbolic Optimal Assembly Program) was an assembly language for the [IBM 650](https://en.wikipedia.org/wiki/IBM_650) computer written by Stan Poley in 1955.[[31]](https://en.wikipedia.org/wiki/Assembly_language#cite_note-32)

Assembly languages eliminate much of the error-prone, tedious, and time-consuming [first-generation](https://en.wikipedia.org/wiki/First-generation_language) programming needed with the earliest computers, freeing programmers from tedium such as remembering numeric codes and calculating addresses. They were once widely used for all sorts of programming. However, by the 1980s (1990s on [microcomputers](https://en.wikipedia.org/wiki/Microcomputer)), their use had largely been supplanted by higher-level languages, in the search for improved [programming productivity](https://en.wikipedia.org/wiki/Programming_productivity). Today assembly language is still used for direct hardware manipulation, access to specialized processor instructions, or to address critical performance issues. Typical uses are [device drivers](https://en.wikipedia.org/wiki/Device_driver), low-level [embedded systems](https://en.wikipedia.org/wiki/Embedded_system), and [real-time](https://en.wikipedia.org/wiki/Real-time_computing) systems.

Historically, numerous programs have been written entirely in assembly language. The [Burroughs MCP](https://en.wikipedia.org/wiki/Burroughs_MCP) (1961) was the first computer for which an operating system was not developed entirely in assembly language; it was written in [Executive Systems Problem Oriented Language](https://en.wikipedia.org/wiki/Executive_Systems_Problem_Oriented_Language) (ESPOL), an Algol dialect. Many commercial applications were written in assembly language as well, including a large amount of the [IBM mainframe](https://en.wikipedia.org/wiki/IBM_mainframe) software written by large corporations. [COBOL](https://en.wikipedia.org/wiki/COBOL), [FORTRAN](https://en.wikipedia.org/wiki/FORTRAN) and some [PL/I](https://en.wikipedia.org/wiki/PL/I) eventually displaced much of this work, although a number of large organizations retained assembly-language application infrastructures well into the 1990s.

Most early microcomputers relied on hand-coded assembly language, including most operating systems and large applications. This was because these systems had severe resource constraints, imposed idiosyncratic memory and display architectures, and provided limited, buggy system services. Perhaps more important was the lack of first-class high-level language compilers suitable for microcomputer use. A psychological factor may have also played a role: the first generation of microcomputer programmers retained a hobbyist, "wires and pliers" attitude.

In a more commercial context, the biggest reasons for using assembly language were minimal bloat (size), minimal overhead, greater speed, and reliability.

Typical examples of large assembly language programs from this time are IBM PC [DOS](https://en.wikipedia.org/wiki/DOS) operating systems, the [Turbo Pascal](https://en.wikipedia.org/wiki/Turbo_Pascal) compiler and early applications such as the [spreadsheet](https://en.wikipedia.org/wiki/Spreadsheet) program [Lotus 1-2-3](https://en.wikipedia.org/wiki/Lotus_1-2-3). According to some[[*who?*](https://en.wikipedia.org/wiki/Wikipedia:Manual_of_Style/Words_to_watch#Unsupported_attributions)] industry insiders, the assembly language was the best computer language to use to get the best performance out of the [Sega Saturn](https://en.wikipedia.org/wiki/Sega_Saturn), a console that was notoriously challenging to develop and program games for.[[32]](https://en.wikipedia.org/wiki/Assembly_language#cite_note-33) The 1993 arcade game [*NBA Jam*](https://en.wikipedia.org/wiki/NBA_Jam_(1993_video_game)) is another example.

Assembly language has long been the primary development language for many popular home computers of the 1980s and 1990s (such as the [MSX](https://en.wikipedia.org/wiki/MSX), [Sinclair](https://en.wikipedia.org/wiki/Sinclair_Research) [ZX Spectrum](https://en.wikipedia.org/wiki/ZX_Spectrum), [Commodore 64](https://en.wikipedia.org/wiki/Commodore_64), [Commodore Amiga](https://en.wikipedia.org/wiki/Commodore_Amiga), and [Atari ST](https://en.wikipedia.org/wiki/Atari_ST)). This was in large part because [interpreted](https://en.wikipedia.org/wiki/Interpreted_language) BASIC dialects on these systems offered insufficient execution speed, as well as insufficient facilities to take full advantage of the available hardware on these systems. Some systems even have an [integrated development environment](https://en.wikipedia.org/wiki/Integrated_development_environment) (IDE) with highly advanced debugging and macro facilities. Some compilers available for the [Radio Shack](https://en.wikipedia.org/wiki/Radio_Shack) [TRS-80](https://en.wikipedia.org/wiki/TRS-80) and its successors had the capability to combine inline assembly source with high-level program statements. Upon compilation a built-in assembler produced inline machine code.

**Current usage**

There have always been debates over the usefulness and performance of assembly language relative to high-level languages. Assembly language has specific niche uses where it is important; see below. As of July 2017, the [TIOBE index](https://en.wikipedia.org/wiki/TIOBE_index) of programming language popularity ranks assembly language at 11, ahead of [Visual Basic](https://en.wikipedia.org/wiki/Visual_Basic), for example.[[33]](https://en.wikipedia.org/wiki/Assembly_language#cite_note-tiobe-34) Assembler can be used to optimize for speed or optimize for size. In the case of speed optimization, modern [optimizing compilers](https://en.wikipedia.org/wiki/Optimizing_compiler) are claimed[[34]](https://en.wikipedia.org/wiki/Assembly_language#cite_note-35) to render high-level languages into code that can run as fast as hand-written assembly, despite the counter-examples that can be found.[[35]](https://en.wikipedia.org/wiki/Assembly_language#cite_note-goto-36)[[36]](https://en.wikipedia.org/wiki/Assembly_language#cite_note-bit-fild-37)[[37]](https://en.wikipedia.org/wiki/Assembly_language#cite_note-gcc-mess-38) The complexity of modern processors and memory sub-systems makes effective optimization increasingly difficult for compilers, as well as assembly programmers.[[38]](https://en.wikipedia.org/wiki/Assembly_language#cite_note-GreatDebate1-39)[[39]](https://en.wikipedia.org/wiki/Assembly_language#cite_note-compiler-fails1-40) Moreover, increasing processor performance has meant that most CPUs sit idle most of the time,[[40]](https://en.wikipedia.org/wiki/Assembly_language#cite_note-41) with delays caused by predictable bottlenecks such as cache misses, [I/O](https://en.wikipedia.org/wiki/I/O) operations and [paging](https://en.wikipedia.org/wiki/Paging). This has made raw code execution speed a non-issue for many programmers.

There are some situations in which developers might choose to use assembly language:

* A stand-alone executable of compact size is required that must execute without recourse to the [run-time](https://en.wikipedia.org/wiki/Run-time_system) components or [libraries](https://en.wikipedia.org/wiki/Library_(computing)) associated with a high-level language; this is perhaps the most common situation. For example, firmware for telephones, automobile fuel and ignition systems, air-conditioning control systems, security systems, and sensors.
* Code that must interact directly with the hardware, for example in [device drivers](https://en.wikipedia.org/wiki/Device_driver) and [interrupt handlers](https://en.wikipedia.org/wiki/Interrupt_handler).
* In an embedded processor or DSP, high-repetition interrupts require the shortest number of cycles per interrupt, such as an interrupt that occurs 1000 or 10000 times a second.
* Programs that need to use processor-specific instructions not implemented in a compiler. A common example is the [bitwise rotation](https://en.wikipedia.org/wiki/Circular_shift) instruction at the core of many encryption algorithms, as well as querying the parity of a byte or the 4-bit carry of an addition.
* Programs that create vectorized functions for programs in higher-level languages such as C. In the higher-level language this is sometimes aided by compiler [intrinsic functions](https://en.wikipedia.org/wiki/Intrinsic_function)which map directly to SIMD mnemonics, but nevertheless result in a one-to-one assembly conversion specific for the given vector processor.
* Programs requiring extreme optimization, for example an inner [loop](https://en.wikipedia.org/wiki/Program_loop) in a processor-intensive algorithm. [Game programmers](https://en.wikipedia.org/wiki/Game_programmer) take advantage of the abilities of hardware features in systems, enabling games to run faster. Also large scientific simulations require highly optimized algorithms, e.g. [linear algebra](https://en.wikipedia.org/wiki/Linear_algebra) with [BLAS](https://en.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms)[[35]](https://en.wikipedia.org/wiki/Assembly_language#cite_note-goto-36)[[41]](https://en.wikipedia.org/wiki/Assembly_language#cite_note-bench-42) or [discrete cosine transformation](https://en.wikipedia.org/wiki/DCT_(math))(e.g. [SIMD](https://en.wikipedia.org/wiki/SIMD) assembly version from [x264](https://en.wikipedia.org/wiki/X264)[[42]](https://en.wikipedia.org/wiki/Assembly_language#cite_note-43))
* Situations where no high-level language exists, on a new or specialized processor, for example.
* Programs that need precise timing such as
  + [real-time](https://en.wikipedia.org/wiki/Real-time_computing) programs such as simulations, flight navigation systems, and medical equipment. For example, in a [fly-by-wire](https://en.wikipedia.org/wiki/Fly-by-wire) system, telemetry must be interpreted and acted upon within strict time constraints. Such systems must eliminate sources of unpredictable delays, which may be created by (some) interpreted languages, automatic [garbage collection](https://en.wikipedia.org/wiki/Garbage_collection_(computer_science)), paging operations, or [preemptive multitasking](https://en.wikipedia.org/wiki/Preemptive_multitasking). However, some higher-level languages incorporate run-time components and operating system interfaces that can introduce such delays. Assembly language is still taught in most [computer science](https://en.wikipedia.org/wiki/Computer_science) and [electronic engineering](https://en.wikipedia.org/wiki/Electronic_engineering) programs. Although few programmers today regularly work with assembly language as a tool, the underlying concepts remain very important. Such fundamental topics as [binary arithmetic](https://en.wikipedia.org/wiki/Binary_arithmetic), [memory allocation](https://en.wikipedia.org/wiki/Memory_allocation), [stack processing](https://en.wikipedia.org/wiki/Stack_(data_structure)), [character set](https://en.wikipedia.org/wiki/Character_set) encoding, [interrupt](https://en.wikipedia.org/wiki/Interrupt) processing, and [compiler](https://en.wikipedia.org/wiki/Compiler) design would be hard to study in detail without a grasp of how a computer operates at the hardware level. Since a computer's behavior is fundamentally defined by its instruction set, the logical way to learn such concepts is to study an assembly language. Most modern computers have similar instruction sets. Therefore, studying a single assembly language is sufficient to learn: I) the basic concepts; II) to recognize situations where the use of assembly language might be appropriate; and III) to see how efficient executable code can be created from high-level languages.[[45]](https://en.wikipedia.org/wiki/Assembly_language#cite_note-46) This is analogous to children needing to learn the basic arithmetic operations (e.g., long division), although [calculators](https://en.wikipedia.org/wiki/Calculator) are widely used for all except the most trivial calculations.