Assemblers

6502 assemblers

Assembler	License	Instruction set	Host platform
64tass	GPL	MOS Technology 6502, WDC 65C02, WDC 65816/65802	various
ACME	GPL	MOS Technology 6502, WDC 65C02, WDC 65816/65802	various
ASM6	Public domain	MOS Technology 6502	various
ATASM	GPL	MOS Technology 6502	various
Atari Assembler Editor	Proprietary	MOS Technology 6502	Atari 8-bit family
C64List	Proprietary	MOS Technology 6502	Commodore 64
CA65	GPL	MOS Technology 6502, WDC 65C02, WDC 65816/65802	various
dasm	GPL	MOS Technology 6502, others	various
dreamass	GPL	MOS Technology 6502, WDC 65816/65802	various
French Silk	Proprietary	MOS Technology 6502	Commodore 64
Kick Assembler	Proprietary	MOS Technology 6502	various
Lisa	Proprietary	MOS Technology 6502	Apple II series
MAC/65	Proprietary	MOS Technology 6502	Atari 8-bit family

Assembler	License	Instruction set	Host platform
Merlin	Proprietary	MOS Technology 6502, WDC 65C02, WDC 65816/65802	Apple II series, Commodore 64, Commodore 128
WLA DX	GPL	MOS Technology 6502, others	various
XA65	GPL	MOS Technology 6502, others	various
XASM	Public domain	MOS Technology 6502	various

680×0 assemblers

Assembler	License	Instruction set	Host platform
A68K	Free	Motorola 680×0	Commodore Amiga
ASM-One Macro Assembler	Free	Motorola 680×0	Commodore Amiga
Digital Research Assembler	Proprietary	Motorola 680×0	Atari ST
Fantasm	Proprietary	Motorola 680×0	Apple Macintosh
GFA-Assembler	Proprietary	Motorola 680×0	Atari ST
GST Macro Assembler	Proprietary	Motorola 680×0	Atari ST
HiSoft DevPac Assembler	Proprietary	Motorola 680×0	Commodore Amiga, Atari ST
Mac Assembler	Proprietary	Motorola 680×0	Apple Macintosh
MaxonASM	Proprietary	Motorola 680×0	Commodore Amiga

Assembler	License	Instruction set	Host platform
Metacomco Macro Assembler	Proprietary	Motorola 680×0	Commodore Amiga, Atari ST
MPW Assembler	Proprietary	Motorola 680×0	Apple Macintosh
OMA	Proprietary	Motorola 680×0	Commodore Amiga
PhxAss	Free	Motorola 680×0	Commodore Amiga
Seka Assembler	Proprietary	Motorola 680×0	Commodore Amiga, Atari ST

ARM assemblers

Assembler	License	Instruction set	Host platform
Archimedes Assembler	Proprietary	ARM	Acorn Archimedes
ARM, inc. armasm	Proprietary	ARM	Linux, Windows
FASMARM	Free	ARM	various
IAR ARM Assembler	Proprietary	ARM	Windows
Microsoft armasm	Proprietary	ARM	Visual Studio 2005

IBM mainframe assemblers

Assembler	License	Instruction set	Host platform
BAL	Free	IBM System/360	IBM BPS/360
Dignus Systems/ASM	Proprietary	z/Architecture	numerous
HLASM	Proprietary	z/Architecture	z/Architecture
IBM Assembler XF	Proprietary	IBM System/370	IBM System/370

Assembler	License	Instruction set	Host platform
PL360	Free	IBM System/360	IBM System/360

Power Architecture assemblers

Assembler	License	Instruction set	Host platform
IBM AIX assembler	Proprietary	POWER	IBM AIX
MPW Power Assembler	Proprietary	PowerPC	Apple Power Macintosh
Power Fantasm	Proprietary	PowerPC	Apple Power Macintosh
StormPowerASM	Proprietary	PowerPC	PowerPC Amiga

x86 assemblers

Assembler	OS	Open source	License	x86-64	Active Development
A86/A386	Windows, DOS	No	Proprietary	No	No
ACK	Linux, Minix, Unix-like	Yes	BSD since 2003	No	1985-*?*[1]
Arrowsoft Assembler	DOS	No	Public Domain	No	No
IBM ALP	OS/2	No	Proprietary	No	No
AT&T	Unix System V	No	Proprietary	No	1985-?[2]
Bruce D. Evans' as86	Minix 1.x, 16-bit part in Linux	Yes	GPL	No	1988-2001[3]

Assembler	OS	Open source	License	x86-64	Active Development
Digital Research ASM86	CP/M-86, DOS, Intel's ISIS and iRMX	No	Proprietary	No	1978-1992
DevelSoftware Assembler	Windows, Linux, Unix-like	No	Free	Listed, N/A	No
FASM	Windows, DOS, Linux, Unix-like	Yes	BSD with added Copyleft	Yes	Yes
GAS	Unix-like, Windows, DOS, OS/2	Yes	GPL	Yes	Since 1987
GoAsm	Windows	No	Free	Yes	Yes
HLA	Windows, Linux, FreeBSD, Mac OS X	Yes	Public domain	No	Yes
JWASM	Windows, DOS, Linux, FreeBSD, OS/2	Yes	Sybase Open Watcom Public License	Yes	Yes
LZASM	Windows, DOS	No	Free	No	No
MASM	Windows, DOS, OS/2	No	Microsoft EULA	Yes	Since 1981[4]
Mical a86	Unix, DOS, PC/IX	Yes	*?*	No	1982-1984[5]

Assembler	OS	Open source	License	x86-64	Active Development
NASM	Windows, Linux, Mac OS X, DOS, OS/2	Yes	BSD	Yes	Yes
Tim Paterson's ASM	86-DOS, DOS DEBUG	No	Proprietary	No	1979-1983
POASM	Windows, Windows Mobile	No	Free	Yes	Yes
RosAsm	Windows	Yes	GPL	No	No[6]
SLR's OPTASM	DOS	No	Proprietary	No	No
TASM	Windows, DOS	No	Proprietary	No	*?*[7][8]
WASM	Windows, DOS, OS/2	Yes	Sybase Open Watcom Public License	No	*?*
TCCASM	Unix-like, Windows	Yes	LGPL	Yes	Yes
Xenix	Xenix 2.3 and 3.0 (before 1985)	No	Proprietary	No	1982-1984
Yasm	Windows, DOS, Linux, Unix-like	Yes	BSD	Yes	Yes

1. ^ Part of the Minix 3 source tree, but without obvious development activity. The full source history is available.
2. ^ Developed by Interactive in 1986 when they ported

System V to Intel iAPX286 and 80386 architectures. Archetypical of ATT syntax because it was used as reference for GAS. Still used for The SCO Group's products, Unixware and OpenServer.

3. ^ Home site does not appear active any more. Also offered as part of FreeBSD Ports, in bcc-1995.03.12.
4. ^ Active and supported, but not advertised.
5. ^ Developed in 1982 at MIT as a cross-assembler, it was picked up by Interactive in 1983 when they developed PC/IX under IBM contract. The syntax was later used as base for ACK assembler, to be used in Minix 1.x toolchain.
6. ^ RosAsm project on WebArchive.org.
7. ^ Part of the C++Builder Tool Chain, but not sold as a stand-alone product, or marketed since the CodeGear spin-off; Borland was still selling it until then. Version 5.0, the last, is dated 1996.
8. ^ Turbo Assembler was developed as "Turbo Editasm" by Uriah Barnett from Speedware Inc (Sacramento, CA) between 1984 and 1987. It was later sold to (or marketed by) Borland as their Turbo Assembler.

Other architectures

Assembler	License	Instruction set	Host platform
ALM (Assembly Language for Multics)	MIT License	GE-645 Honeywell 6180	GE-645 Honeywell 6180
Babbage	Proprietary	GEC 4000 series	GEC 4000 series
COMPASS[1]	Proprietary	CDC mainframe	CDC mainframe
MACRO-10	Free	PDP-10	PDP-10
MACRO-11	Unknown	PDP-11	PDP-11
MACRO-32	Unknown	VAX	VAX
PASMO	GPL	Zilog Z80	numerous

Assembler	License	Instruction set	Host platform
MRS	GPL	Zilog Z80, 8080	ZX Spectrum, PMD-85
ASEM-51	Free	8051	Embedded Systems
GPASM	GPL	PIC microcontroller	many
ID3E	Free for academic use	SC123	SC123 emulator
MIPS	Free	MIPS architecture	MIPS architecture
SOAP (Symbolic Optimal Assembly Program)	Proprietary	IBM 650	IBM 650
MPW IIgs Assembler	Proprietary	WD 65C816	Apple IIgs
MetaSymbol	Free	SDS/XDS Sigma systems	SDS/XDS Sigma systems
Autocoder[2]	Free	IBM 705, 14xx, 1410, 7010, 7070, 7072, 7074, 7080	various
FAP (Fortran Assembly Program)	Free	IBM 709, 704x, 709x	various
MAP (Macro Assembly Program)	Free	IBM 709, 704x, 709x	various
Symbolic Programming System (SPS)[3]	Free	IBM 14xx, 1620, 1710	IBM 1401, 1440, 1460, 1620, 1710

Disassembler

A **disassembler** is a computer program that translates machine language into assembly language—the inverse operation to that of an assembler. A disassembler differs from a decompiler, which targets a high-level language rather than an assembly language. Disassembly, the output of a disassembler, is often formatted for human-readability rather than suitability for input to an assembler, making it principally a reverse-engineering tool.

Assembly language source code generally permits the use of constants and programmer comments. These are usually removed from the assembled machine code by the assembler. If so, a disassembler operating on the machine code would produce disassembly lacking these constants and comments; the disassembled output becomes more difficult for a human to interpret than the original annotated source code. Some disassemblers make use of the symbolic debugging information present in object files such as ELF. The Interactive Disassembler allow the human user to make up mnemonic symbols for values or regions of code in an interactive session: human insight applied to the disassembly process often parallels human creativity in the code writing process.

Compiler

A **compiler** is a computer program (or set of programs) that transforms source code written in a programming language (the source language) into another computer language (the target language, often having a binary form known as object code).The most common reason for wanting to transform source code is to create an executable program.

The name “compiler” is primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g., assembly language or machine code). If the compiled program can run on a computer whose CPU or operating system is different from the one on which the compiler runs, the compiler is known as a cross-compiler. A program that translates from a low level language to a higher level one is a decompiler. A program that translates between high-level languages is usually called a language translator, source to source translator, or language converter. A language rewriter is usually a program that translates the form of expressions without a change of language.

A compiler is likely to perform many or all of the following operations: lexical analysis, preprocessing, parsing, semantic analysis (Syntax-directed translation), code generation, and code optimization.

Program faults caused by incorrect compiler behavior can be very difficult to track down and work around; therefore, compiler implementors invest significant effort to ensure compiler correctness.

Macros

Macros

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. 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).

Note that this definition of “macro” is slightly different from the use of the term in other contexts, like the C programming language. C macros created through the #define directive typically are just one line, or a few lines at most. Assembler macro instructions can be lengthy “programs” by themselves, executed by interpretation by the assembler during assembly.

Since macros can have ‘short’ names but expand to several or indeed many lines of code, they can be used to make assembly language programs appear to be far shorter, requiring fewer lines of source code, as with higher level languages. They can also be used to add higher levels of structure to assembly programs, optionally introduce embedded debugging code via parameters and other similar features.

Macro assemblers often allow macros to take parameters. Some assemblers include quite sophisticated macro languages, incorporating such high-level language elements as optional parameters, symbolic variables, conditionals, string

manipulation, and arithmetic operations, all usable during the execution of a given macro, and allowing macros to save context or exchange information. Thus a macro might generate a large number of assembly language instructions or data definitions, based on the macro arguments. This could be used to generate record-style data structures or “unrolled” loops, for example, or could generate entire algorithms based on complex parameters. An organization using assembly language that has been heavily extended using such a macro suite can be considered to be working in a higher-level language, since such programmers are not working with a computer’s lowest-level conceptual elements.

Macros were used to customize large scale software systems for specific customers in the mainframe era and were also used by customer personnel to satisfy their employers’ needs by making specific versions of manufacturer operating systems. This was done, for example, by systems programmers working with IBM’s Conversational Monitor System / Virtual Machine (VM/CMS) and with IBM’s “real time transaction processing” add-ons, Customer Information Control System CICS, and ACP/TPF, the airline/financial system that began in the 1970s and still runs many large computer reservations systems (CRS) and credit card systems today.

It was also possible to use solely the macro processing abilities of an assembler to generate code written in completely different languages, for example, to generate a version of a program in COBOL using a pure macro assembler program containing lines of COBOL code inside assembly time operators instructing the assembler to generate arbitrary code.

This was because, as was realized in the 1960s, the concept of “macro processing” is independent of the concept of “assembly”, the former being in modern terms more word processing, text processing, than generating object code. The concept of macro processing appeared, and appears, in the C

programming language, which supports “preprocessor instructions” to set variables, and make conditional tests on their values. Note that unlike certain previous macro processors inside assemblers, the C preprocessor was not Turing-complete because it lacked the ability to either loop or “go to”, the latter allowing programs to loop.

Despite the power of macro processing, it fell into disuse in many high level languages (major exceptions being C/C++ and PL/I) while remaining a perennial for assemblers.

Macro parameter substitution is strictly by name: at macro processing time, the value of a parameter is textually substituted for its name. The most famous class of bugs resulting was the use of a parameter that itself was an expression and not a simple name when the macro writer expected a name. In the macro: foo: macro a load a*b the intention was that the caller would provide the name of a variable, and the “global” variable or constant b would be used to multiply “a”. If foo is called with the parameter a-c, the macro expansion of load a-c*b occurs. To avoid any possible ambiguity, users of macro processors can parenthesize formal parameters inside macro definitions, or callers can parenthesize the input parameters.

Assembly directives

Assembly directives

Assembly directives, also called pseudo-opcodes, pseudo-operations or pseudo-ops, are instructions that are executed by an assembler at assembly time, not by a CPU at run time.

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 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. 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 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, automatically calculate offsets within data structures, 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 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. It should be noted that the “raw” (uncommented) assembly language generated by compilers or disassemblers is quite difficult to read when changes must be made.

Assembly language

Assembly language

A program written in assembly language consists of a series of (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 mnemonic followed by a list of data, arguments or parameters.[4] These are translated by an assembler into machine language instructions that can be loaded into memory and executed.

For example, the instruction below tells an x86/IA-32 processor to move an immediate 8-bit value into a 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 loads the *AL* register with the data 01100001.

10110000 01100001

This binary computer code can be made more human-readable by expressing it in hexadecimal as follows.

B0 61

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. Intel assembly language provides the mnemonic MOV (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.

```
MOV AL, 61h  ; Load AL with 97 decimal (61 hex)
```

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 opcodes 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 Intel 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.

```
MOV AL, 1h       ; Load AL with immediate value 1
MOV CL, 2h       ; Load CL with immediate value 2
MOV DL, 3h       ; Load DL with immediate value 3
```

The syntax of MOV can also be more complex as the following examples show.

```
MOV EAX, [EBX]   ; Move the 4 bytes in memory at the address contained in EBX into EAX
MOV [ESI+EAX], CL ; Move the contents of CL into the byte at address ESI+EAX
```

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.

Transforming assembly language into machine code is the job of an assembler, and the reverse can at least partially be

achieved by a disassembler. Unlike high-level languages, there is usually a one-to-one correspondence between 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 language (discussed below) which is used by vendors and programmers to generate more complex code and data sequences.

Each 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 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.

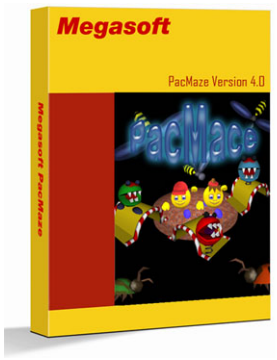
Assembler

Assembler

An **assembler** creates object code by translating assembly instruction mnemonics into opcodes, and by resolving 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.

Assemblers have been available since the 1950s and are far simpler to write than compilers for high-level languages as each mnemonic instruction / address mode combination translates directly into a single machine language opcode. Modern assemblers, especially for RISC architectures, such as SPARC or Power Architecture, as well as x86 and x86-64, optimize instruction scheduling to exploit the CPU pipeline efficiently.

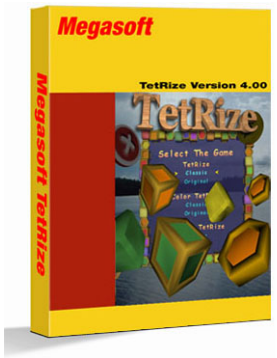
New games



PacMaze, Pacman-like arcade game.

The full version offers 80 mazes with power shields, freeze bonuses, magic bridges and other features.

FREE trial version of PacMaze: Download Now!



TetRize/Color TetRize/Colorix TetRize

Rules are the same as in Classic, but game includes new bonuses. Sometimes throughout the game bonuses appeares in the playing area.

FREE trial version of PacMaze: Download Now!