Assembly language

In computer programming, **assembly language** (or **assembler language**),[1] sometimes 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.[2] 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*.[3] [4]

Assembly language

```
MONITOR FOR 6802 1.4          9-14-80  TSC ASSEMBLER  PAGE    2

 C000                   ORG    ROM+$0000 BEGIN MONITOR
 C000 8E 00 70  START   LDS    #STACK

                ************************************
                * FUNCTION: INITA - Initialize ACIA
                * INPUT: none
                * OUTPUT: none
                * CALLS: none
                * DESTROYS: acc A

 0013           RESETA  EQU    %00010011
 0011           CTLREG  EQU    %00010001

 C003 86 13     INITA   LDA A  #RESETA   RESET ACIA
 C005 B7 80 04          STA A  ACIA
 C008 86 11             LDA A  #CTLREG   SET 8 BITS AND 2 STOP
 C00A B7 80 04          STA A  ACIA

 C00D 7E C0 F1          JMP    SIGNON    GO TO START OF MONITOR

                ************************************
                * FUNCTION: INCH - Input character
                * INPUT: none
                * OUTPUT: char in acc A
                * DESTROYS: acc A
                * CALLS: none
                * DESCRIPTION: Gets 1 character from terminal

 C010 B6 80 04  INCH    LDA A  ACIA      GET STATUS
 C013 47                ASR A            SHIFT RDRF FLAG INTO CARRY
 C014 24 FA             BCC    INCH      RECIEVE NOT READY
 C016 B6 80 05          LDA A  ACIA+1    GET CHAR
 C019 84 7F             AND A  #$7F      MASK PARITY
 C01B 7E C0 79          JMP    OUTCH     ECHO & RTS

                ************************************
                * FUNCTION: INHEX - INPUT HEX DIGIT
                * INPUT: none
                * OUTPUT: Digit in acc A
                * CALLS: INCH
                * DESTROYS: acc A
                * Returns to monitor if not HEX input

 C01E 8D F0     INHEX   BSR    INCH      GET A CHAR
 C020 81 30             CMP A  #'0       ZERO
 C022 2B 11             BMI    HEXERR    NOT HEX
 C024 81 39             CMP A  #'9       NINE
 C026 2F 0A             BLE    HEXRTS    GOOD HEX
 C028 81 41             CMP A  #'A
 C02A 2B 09             BMI    HEXERR    NOT HEX
 C02C 81 46             CMP A  #'F
 C02E 2E 05             BGT    HEXERR
 C030 80 07             SUB A  #7        FIX A-F
 C032 84 0F     HEXRTS  AND A  #$0F      CONVERT ASCII TO DIGIT
 C034 39                RTS

 C035 7E C0 AF  HEXERR  JMP    CTRL      RETURN TO CONTROL LOOP
```

Typical *secondary output* from an assembler—showing original assembly language (right) for the Motorola MC6800 and the assembled form

Paradigm	Imperative, unstructured
First appeared	1949; 72 years ago

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,[5] macros,[6][1] and symbolic labels of program and memory locations are often also supported.

The term "assembler" is generally attributed to Wilkes, Wheeler and Gill in their 1951 book *The Preparation of Programs for an Electronic Digital Computer*,[7] who, however, used the term to mean "a program that assembles another program consisting of several sections into a single program".[8]

Each assembly language is specific to a particular computer architecture and sometimes to an operating system.[9] However, some assembly languages do not provide specific syntax for operating system calls, and most assembly languages can be used universally with any operating system, as the language provides access to all the real capabilities of the processor, upon which all system call mechanisms ultimately rest. In contrast to assembly languages, most high-level programming languages are generally portable across multiple architectures but require interpreting or compiling, a much more complicated task than assembling.

The computational step when an assembler is processing a program is called *assembly time*.

Assembly language syntax

Assembly language uses a mnemonic to represent each low-level machine instruction or opcode, typically also each architectural register, flag, etc. Many operations require one or more operands in order to form a complete instruction. Most assemblers permit named constants, registers, and labels for program and memory locations, and can calculate expressions for operands. Thus, 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 using offsets 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.

Terminology

- A **macro assembler** includes a macroinstruction 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) is an assembler that is run on a computer or 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 or a microcontroller. In such a case, the resulting object code must be transferred to the target system, via read-only memory (ROM, EPROM, etc.), a programmer (when the read-only memory is integrated in the device, as in microcontrollers), or a data link using either an exact bit-by-bit copy of the object code or a text-based representation of that code (such as Intel hex or Motorola S-record).
- A **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, DO CASE, etc.) and high-level abstract data types, including structures/records, unions, classes, and sets.
- A **microassembler** is a program that helps prepare a microprogram, called *firmware*, to control the low level operation of a computer.
- A **meta-assembler** is "a program that accepts the syntactic and semantic description of an assembly language, and generates an assembler for that language".[10] "Meta-Symbol" assemblers for the SDS 9 Series and SDS Sigma series of computers are meta-assemblers.[11][nb 1]Sperry Univac also provided a Meta-Assembler for the UNIVAC 1100/2200 series.[12]
- **inline assembler** (or **embedded assembler**) is assembler code contained within a high-level language program.[13] This is most often used in systems programs which need direct access to the hardware.

Key concepts

Assembler

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.[14] 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,[14] 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.[*citation needed*]

Assemblers have been available since the 1950s, as the first step above machine language and before high-level programming languages such as Fortran, Algol, COBOL and Lisp. There have also been several classes of translators and semi-automatic code generators with properties similar to both assembly and high-level languages, with Speedcode as perhaps one of the better-known examples.

There may be several assemblers with different syntax for a particular CPU or instruction set architecture. For instance, an instruction to add memory data to a register in a x86-family processor might be `add eax, [ebx]`, in original *Intel syntax*, whereas this would be written `addl (%ebx),%eax` in the *AT&T syntax* used by the GNU Assembler. Despite different appearances, different syntactic forms generally generate the same numeric machine code. 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-syntax, TASM-syntax, ideal mode, etc., in the special case of x86 assembly 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" 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. [15]

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.

```
S1    B     FWD
  ...
FWD   EQU *
  ...
BKWD  EQU *
  ...
S2    B     BKWD
```

High-level assemblers

More sophisticated high-level assemblers 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, e.g., the IBM 700 series and IBM 7000 series, and since the 1960s for IBM System/360 (S/360), amongst other machines)
- Object-oriented programming features such as classes, objects, abstraction, polymorphism, and inheritance[16]

See Language design below for more details.

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 an operand, which might be a list of data, arguments or parameters.[17] Some instructions may be "implied," which means the data upon which the instruction operates is implicitly defined by the instruction itself—such an instruction does not take an operand. The resulting statement is 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.[17]

```
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. Assembly language for the 8086 family 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 (including this one) 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 or by immediate (a.k.a direct) addresses. 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.

If the same mnemonic is used for different instructions, that means that the mnemonic corresponds to several different binary instruction codes, excluding data (e.g. the `61h` in this example), depending on the operands that follow the mnemonic. For example, for the x86/IA-32 CPUs, the Intel assembly language syntax `MOV AL, AH` represents an instruction that moves the contents of register *AH* into register *AL*. The[nb 2] hexadecimal form of this instruction is:

```
88 E0
```

The first byte, 88h, identifies a move between a byte-sized register and either another register or memory, and the second byte, E0h, is encoded (with three bit-fields) to specify that both operands are registers, the source is *AH*, and the destination is *AL*.

In a case like this where the same mnemonic can represent more than one binary instruction, the assembler determines which instruction to generate by examining the operands. In the first example, the operand `61h` is a valid hexadecimal numeric constant and is not a valid register name, so only the `B0` instruction can be applicable. In the second example, the operand `AH` is a valid register name and not a valid numeric constant (hexadecimal, decimal, octal, or binary), so only the `88` instruction can be applicable.

Assembly languages are always designed so that this sort of unambiguousness is universally enforced by their syntax. For example, in the Intel x86 assembly language, a hexadecimal constant must start with a numeral digit, so that the hexadecimal number 'A' (equal to decimal ten) would be written as `0Ah` or `0AH`, not `AH`, specifically so that it cannot appear to be the name of register *AH*. (The same rule also prevents ambiguity with the names of registers *BH*, *CH*, and *DH*, as well as with any user-defined symbol that ends with the letter *H* and otherwise contains only characters that are hexadecimal digits, such as the word "BEACH".)

Returning to the original example, while the x86 opcode 10110000 (`B0`) copies an 8-bit value into the *AL* register, 10110001 (`B1`) moves it into *CL* and 10110010 (`B2`) does so into *DL*. Assembly language examples for these follow.[17]

```
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.[18]

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

In each case, the MOV mnemonic is translated directly into one of the opcodes 88-8C, 8E, A0-A3, B0-BF, C6 or C7 by an assembler, and the programmer normally does not have to know or remember which.[17]

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 a 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 language (discussed below) which is used by vendors and programmers to generate more complex code and data sequences. Since the information about pseudoinstructions and macros defined in the assembler environment is not present in the object program, a disassembler cannot reconstruct the macro and pseudoinstruction invocations but can only disassemble the actual machine instructions that the assembler generated from those abstract assembly-language entities. Likewise, since comments in the assembly language source file are ignored by the assembler and have no effect on the object code it generates, a disassembler is always completely unable to recover source comments.

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 CPU manufacturer and used in its documentation.

Two examples of CPUs that have two different sets of mnemonics are the Intel 8080 family and the Intel 8086/8088. Because Intel claimed copyright on its assembly language mnemonics (on each page of their documentation published in the 1970s and early 1980s, at least), some companies that independently produced CPUs compatible with Intel instruction sets invented their own mnemonics. The Zilog Z80 CPU, an enhancement of the Intel 8080A, supports all the 8080A instructions plus many more; Zilog invented an entirely new assembly language, not only for the new instructions but also for all of the 8080A instructions. For example, where Intel uses the mnemonics *MOV*, *MVI*, *LDA*, *STA*, *LXI*, *LDAX*, *STAX*, *LHLD*, and *SHLD* for various data transfer instructions, the Z80 assembly language uses the mnemonic *LD* for all of them. A similar case is the NEC V20 and V30 CPUs, enhanced copies of the Intel 8086 and 8088, respectively. Like Zilog with the Z80, NEC invented new mnemonics for all of the 8086 and 8088 instructions, to avoid accusations of infringement of Intel's copyright. (It is questionable whether such copyrights can be valid, and later CPU companies such as AMD[nb 3] and Cyrix republished Intel's x86/IA-32 instruction mnemonics exactly with neither permission nor legal penalty.) It is doubtful whether in practice many people who programmed the V20 and V30 actually wrote in NEC's assembly language rather than Intel's; since any two assembly languages for the same instruction set architecture are isomorphic (somewhat like English and Pig Latin), there is no requirement to use a manufacturer's own published assembly language with that manufacturer's products.

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 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. Generally, a mnemonic is a symbolic name for a single executable machine language instruction (an 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*. 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 and 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*.[19]

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,bc` is recognized to generate `ld l,c` followed by `ld h,b`.[20] These are sometimes known as *pseudo-opcodes*.

Mnemonics are arbitrary symbols; in 1985 the 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 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".[14] 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.[21]

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 in 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 often 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. The "raw" (uncommented) assembly language generated by compilers or disassemblers is quite difficult to read when changes must be made.

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. The macro definition is most commonly[nb 4] 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). Macros in this sense date to IBM autocoders of the 1950s.[22][nb 5]

In assembly language, the term "macro" represents a more comprehensive concept than it does in some other contexts, such as the pre-processor in the C programming language, where its #define directive typically is used to create short single line macros. Assembler macro instructions, like macros in PL/I and some other languages, 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 numerous 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. For instance, a "sort" macro could accept the specification of a complex sort key and generate code crafted for that specific key, not needing the run-time tests that would be required for a general procedure interpreting the specification. 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. Underlining this point, macros were used to implement an early virtual machine in SNOBOL4 (1967), which was written in the SNOBOL Implementation Language (SIL), an assembly language for a virtual machine. The target machine would translate this to its native code using a macro assembler.[23] This allowed a high degree of portability for the time.

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 reservation systems (CRS) and credit card systems today.

It is 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. IBM OS/360 uses macros to perform system generation. The user specifies

options by coding a series of assembler macros. Assembling these macros generates a job stream to build the system, including job control language and utility control statements.

This is 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. Unlike certain previous macro processors inside assemblers, the C preprocessor is not Turing-complete because it lacks 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.[24]

Support for structured programming

Packages of macros have been written providing structured programming elements to encode execution flow. The earliest example of this approach was in the Concept-14 macro set,[25] originally proposed by Harlan Mills (March 1970), and implemented by Marvin Kessler at IBM's Federal Systems Division, which provided IF/ELSE/ENDIF and similar control flow

blocks for OS/360 assembler programs. This was a way to reduce or eliminate the use of GOTO operations in assembly code, one of the main factors causing spaghetti code in assembly language. This approach was widely accepted in the early 1980s (the latter days of large-scale assembly language use).

A curious design was A-natural, a "stream-oriented" assembler for 8080/Z80, processors[*citation needed*] from Whitesmiths Ltd. (developers of the Unix-like Idris operating system, and what was reported to be the first commercial C compiler). The language was classified as an assembler because it worked with raw machine elements such as opcodes, registers, 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.[26] 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.[27]

Assemblers with a strong macro engine allow structured programming via macros, such as the switch macro provided with the Masm32 package (this code is a complete program):

```
include \masm32\include\masm32rt.inc    ; use the Masm32 library

.code
demomain:
  REPEAT 20
    switch rv(nrandom, 9)   ; generate a number between 0 and 8
    mov ecx, 7
    case 0
        print "case 0"
    case ecx                ; in contrast to most other programming languages,
        print "case 7"      ; the Masm32 switch allows "variable cases"
    case 1 .. 3
```

```
        .if eax==1
            print "case 1"
        .elseif eax==2
            print "case 2"
        .else
            print "cases 1 to 3: other"
        .endif
    case 4, 6, 8
        print "cases 4, 6 or 8"
    default
        mov ebx, 19          ; print 20 stars
        .Repeat
            print "*"
            dec ebx
        .Until Sign?         ; loop until the sign flag is set
    endsw
    print chr$(13, 10)
  ENDM
  exit
end demomain
```

Use of assembly language

Historical perspective

Assembly languages were not available at the time when the stored-program computer was introduced. Kathleen Booth "is credited with inventing assembly language"[28][29] based on theoretical work she began in 1947, while working on the ARC2 at Birkbeck, University of London following consultation by Andrew Booth (later her husband) with mathematician John von Neumann and physicist Herman Goldstine at the Institute for Advanced Study.[29][30]

In late 1948, the Electronic Delay Storage Automatic Calculator (EDSAC) had an assembler (named "initial orders") integrated into its bootstrap program. It used one-letter mnemonics developed by David Wheeler, who is credited by the IEEE Computer Society as the creator of the first "assembler".[14][31][32] Reports on the EDSAC introduced the term "assembly" for the process of combining fields into an instruction word.

[33] SOAP (Symbolic Optimal Assembly Program) was an assembly language for the IBM 650 computer written by Stan Poley in 1955.[34]

Assembly languages eliminate much of the error-prone, tedious, and time-consuming first-generation programming needed with the earliest computers, freeing programmers from tedium such as remembering numeric codes and calculating addresses.

Assembly languages were once widely used for all sorts of programming. However, by the 1980s (1990s on microcomputers), their use had largely been supplanted by higher-level languages, in the search for improved 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, low-level embedded systems, and real-time systems.

Historically, numerous programs have been written entirely in assembly language. The 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 (ESPOL), an Algol dialect. Many commercial applications were written in assembly language as well, including a large amount of the IBM mainframe software written by large corporations. COBOL, FORTRAN and some 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 operating systems, the Turbo Pascal compiler and early applications such as the spreadsheet program Lotus 1-2-3. Assembly language was used to get the best performance out of the Sega Saturn, a console that was notoriously challenging to develop and program games for.[35] The 1993 arcade game *NBA Jam* 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, Sinclair ZX Spectrum, Commodore 64, Commodore Amiga, and Atari ST). This was in large part because interpreted 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 (IDE) with highly advanced debugging and macro facilities. Some compilers available for the Radio Shack 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[36] been debates over the usefulness and performance of assembly language relative to high-level languages.

Although assembly language has specific niche uses where it is important (see below), there are other tools for optimization.[37]

As of July 2017, the TIOBE index of programming language popularity ranks assembly language at 11, ahead of Visual Basic, for example.[38] Assembler can be used to optimize for speed or optimize for size. In the case of speed optimization, modern optimizing compilers are claimed[39] to render high-level languages into code that can run as fast as hand-written assembly, despite the counter-examples that can be found.[40][41][42] The complexity of modern processors and memory sub-systems makes effective optimization increasingly difficult for compilers, as well as for assembly programmers.[43][44] Moreover, increasing processor performance has meant that most CPUs sit idle most of the time,[45] with delays caused by predictable bottlenecks such as cache misses, I/O operations and 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:

- Writing code for systems with older processors that have limited high-level language options such as the Atari 2600, Commodore 64, and graphing calculators.[46]
- Code that must interact directly with the hardware, for example in device drivers and interrupt handlers.
- 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 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.
- A stand-alone executable of compact size is required that must execute without recourse to the run-time components or libraries associated with a high-level language. Examples have included firmware for telephones, automobile fuel and ignition systems, air-conditioning control systems, security systems, and sensors.
- Programs with performance-sensitive inner loops, where assembly language provides optimization opportunities that are difficult to achieve in a high-level language. For example, linear algebra with BLAS[40][47] or discrete cosine transformation (e.g. SIMD assembly version from x264[48]).
- 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 which map directly to SIMD mnemonics, but nevertheless result in a one-to-one assembly conversion specific for the given vector processor.
- Real-time programs such as simulations, flight navigation systems, and medical equipment. For example, in a 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, paging operations, or preemptive multitasking. However, some higher-level languages incorporate run-time components and operating system interfaces that can introduce such delays. Choosing assembly or lower level languages

for such systems gives programmers greater visibility and control over processing details.
- Cryptographic algorithms that must always take strictly the same time to execute, preventing timing attacks.
- Modify and extend legacy code written for IBM mainframe computers.[49][50]
- Situations where complete control over the environment is required, in extremely high-security situations where nothing can be taken for granted.
- Computer viruses, bootloaders, certain device drivers, or other items very close to the hardware or low-level operating system.
- Instruction set simulators for monitoring, tracing and debugging where additional overhead is kept to a minimum.
- Situations where no high-level language exists, on a new or specialized processor for which no cross compiler is available.
- Reverse-engineering and modifying program files such as:
  existing binaries that may or may not have originally been written in a high-level language, for example when trying to recreate programs for which source code is not available or has been lost, or cracking copy protection of proprietary software.
  Video games (also termed ROM hacking), which is possible via several methods. The most widely employed method is altering program code at the assembly language level.

Assembly language is still taught in most computer science and electronic engineering programs. Although few programmers today regularly work with assembly language as a tool, the underlying concepts remain important. Such fundamental topics as binary arithmetic, memory allocation, stack processing, character set encoding, interrupt processing, and 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.[16]

Assembly language is also widely used among hobbyists who develop programs for the computers of the 1970s and 1980s often in the context of demoscene or retrogaming subcultures.

Typical applications

- 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.)
- Assembly language is often used for low-level code, for instance for operating system kernels, which cannot rely on the availability of pre-existing system calls and must indeed implement them for the particular processor architecture on which the system will be running.
- Some compilers translate high-level languages into assembly first before fully compiling, allowing the assembly code to be viewed for debugging and optimization purposes.
- 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 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. Tools such as the Interactive Disassembler make extensive use of disassembly for such a purpose. This technique is used by hackers to crack commercial software, and competitors to produce software with similar results from competing companies.
- Assembly language is used to enhance speed of execution, especially in early personal computers with limited processing power and RAM.
- Assemblers can be used to generate blocks of data, with no high-level language overhead, from formatted and commented source code, to be used by other code.[51][52]

See also

- Compiler
- Comparison of assemblers
- Disassembler
- Hexadecimal
- Instruction set architecture
- Little man computer – an educational computer model with a base-10 assembly language
- Nibble
- Typed assembly language

Notes

1. ^ "Used as a meta-assembler, it enables the user to design his own programming languages and to generate processors for such languages with a minimum of effort."
2. ^ This is one of two redundant forms of this instruction that operate identically. The 8086 and several other CPUs from the late 1970s/early 1980s have redundancies in their instruction sets, because it was simpler for engineers to design these CPUs (to fit on silicon chips of limited sizes) with the redundant codes than to eliminate them (see don't-care terms). Each assembler will typically generate only one of two or more redundant instruction encodings, but a *disassembler* will usually recognize any of them.
3. ^ AMD manufactured second-source Intel 8086, 8088, and 80286 CPUs, and perhaps 8080A and/or 8085A CPUs, under license from Intel, but starting with the 80386, Intel refused to share their x86 CPU designs with anyone—AMD sued about this for breach of contract—and AMD designed, made, and sold 32-bit and 64-bit x86-family CPUs without Intel's help or endorsement.
4. ^ In 7070 Autocoder, a macro definition is a 7070 macro generator program that the assembler calls; Autocoder provides special macros for macro generators to use.
5. ^ "The following minor restriction or limitation is in effect with regard to the use of 1401 Autocoder when coding macro instructions ..."

References

1. ^ [a] [b] "Assembler language". *High Level Assembler for z/OS & z/VM & z/VSE Language Reference Version 1 Release 6*. IBM. 2014 [1990]. SC26-4940-06.
2. ^ Saxon, James A.; Plette, William S. (1962). *Programming the IBM 1401, a self-instructional programmed manual*. Englewood Cliffs, New Jersey, USA:

Prentice-Hall. LCCN 62-20615. (NB. Use of the term *assembly program*.)

3. ^ "Assembly: Review" (PDF). Computer Science and Engineering. College of Engineering, The Ohio State University. 2016. Archived (PDF) from the original on 2020-03-24. Retrieved 2020-03-24.
4. ^ Archer, Benjamin (November 2016). *Assembly Language For Students*. North Charleston, South Carolina, USA: CreateSpace Independent Publishing. ISBN 978-1-5403-7071-6. "Assembly language may also be called symbolic machine code."
5. ^ Kornelis, A. F. (2010) [2003]. "High Level Assembler – Opcodes overview, Assembler Directives". Archived from the original on 2020-03-24. Retrieved 2020-03-24.
6. ^ "Macro instructions". *High Level Assembler for z/OS & z/VM & z/VSE Language Reference Version 1 Release 6*. IBM. 2014 [1990]. SC26-4940-06.
7. ^ Wilkes, Maurice Vincent; Wheeler, David John; Gill, Stanley J. (1951). *The preparation of programs for an electronic digital computer* (Reprint 1982 ed.). Tomash Publishers. ISBN 978-0-93822803-5. OCLC 313593586.
8. ^ Fairhead, Harry (2017-11-16). "History of Computer Languages - The Classical Decade, 1950s". *I Programmer*. Archived from the original on 2020-01-02. Retrieved 2020-03-06.
9. ^ "How do assembly languages depend on operating systems?". *Stack Exchange*. Stack Exchange Inc. 2011-07-28. Archived from the original on 2020-03-24. Retrieved 2020-03-24. (NB. System calls often vary, e.g. for MVS vs. VSE vs. VM/CMS; the binary/executable formats for different operating systems may also vary.)
10. ^ Daintith, John, ed. (2019). "meta-assembler". *A Dictionary of Computing*. Archived from the original on 2020-03-24. Retrieved 2020-03-24.
11. ^ Xerox Data Systems (Oct 1975). *Xerox Meta-Symbol Sigma 5-9 Computers Language and Operations Reference Manual* (PDF). p. vi. Retrieved 2020-06-07.
12. ^ Sperry Univac Computer Systems (1977). *Sperry Univac Computer Systems Meta-Assembler (MASM) Programmer Reference* (PDF). Retrieved 2020-06-07.
13. ^ "How to Use Inline Assembly Language in C Code". *gnu.org*. Retrieved 2020-11-05.
14. ^ *a b c d* Salomon, David (February 1993) [1992]. Written at California State University, Northridge, California, USA. Chivers, Ian D. (ed.). *Assemblers and Loaders* (PDF). Ellis Horwood Series In Computers And Their Applications (1 ed.). Chicester, West Sussex, UK: Ellis Horwood Limited / Simon & Schuster International Group. pp. 7, 237–238. ISBN 0-13-052564-2.

Archived (PDF) from the original on 2020-03-23. Retrieved 2008-10-01. (xiv+294+4 pages)

15. ^ Beck, Leland L. (1996). "2". *System Software: An Introduction to Systems Programming*. Addison Wesley.
16. ^ [a] [b] Hyde, Randall (September 2003) [1996-09-30]. "Foreword ("Why would anyone learn this stuff?") / Chapter 12 – Classes and Objects". *The Art of Assembly Language* (2 ed.). No Starch Press. ISBN 1-886411-97-2. Archived from the original on 2010-05-06. Retrieved 2020-06-22. Errata: [1] (928 pages) [2][3]
17. ^ [a] [b] [c] [d] *Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference* (PDF). **2**. Intel Corporation. 1999. Archived from the original (PDF) on 2009-06-11. Retrieved 2010-11-18.
18. ^ Ferrari, Adam; Batson, Alan; Lack, Mike; Jones, Anita (2018-11-19) [Spring 2006]. Evans, David (ed.). "x86 Assembly Guide". Computer Science CS216: Program and Data Representation. University of Virginia. Archived from the original on 2020-03-24. Retrieved 2010-11-18.
19. ^ "The SPARC Architecture Manual, Version 8" (PDF). SPARC International. 1992. Archived from the original (PDF) on 2011-12-10. Retrieved 2011-12-10.
20. ^ Moxham, James (1996). "ZINT Z80 Interpreter". *Z80 Op Codes for ZINT*. Archived from the original on 2020-03-24. Retrieved 2013-07-21.
21. ^ Hyde, Randall. "Chapter 8. MASM: Directives & Pseudo-Opcodes" (PDF). *The Art of Computer Programming*. Archived (PDF) from the original on 2020-03-24. Retrieved 2011-03-19.
22. ^ *Users of 1401 Autocoder*. Archived from the original on 2020-03-24. Retrieved 2020-03-24.
23. ^ Griswold, Ralph E. (1972). "Chapter 1". *The Macro Implementation of SNOBOL4*. San Francisco, California, USA: W. H. Freeman and Company. ISBN 0-7167-0447-1.
24. ^ "Macros (C/C++), MSDN Library for Visual Studio 2008". Microsoft Corp. 2012-11-16. Archived from the original on 2020-03-24. Retrieved 2010-06-22.
25. ^ Kessler, Marvin M. (1970-12-18). "*Concept* Report 14 - Implementation of Macros To Permit Structured Programming in OS/360". *MVS Software: Concept 14 Macros*. Gaithersburg, Maryland, USA: International Business Machines Corporation. Archived from the original on 2020-03-24. Retrieved 2009-05-25.
26. ^ "assembly language: Definition and Much More from Answers.com". *answers.com*. Archived from the original on 2009-06-08. Retrieved 2008-06-19.

27. ^ Provinciano, Brian (2005-04-17). "NESHLA: The High Level, Open Source, 6502 Assembler for the Nintendo Entertainment System". Archived from the original on 2020-03-24. Retrieved 2020-03-24.
28. ^ Dufresne, Steven (2018-08-21). "Kathleen Booth: Assembling Early Computers While Inventing Assembly". Archived from the original on 2020-03-24. Retrieved 2019-02-10.
29. ^ [a] [b] Booth, Andrew Donald; Britten, Kathleen Hylda Valerie (September 1947) [August 1947]. *General considerations in the design of an all purpose electronic digital computer* (PDF) (2 ed.). The Institute for Advanced Study, Princeton, New Jersey, USA: Birkbeck College, London. Archived (PDF) from the original on 2020-03-24. Retrieved 2019-02-10. "The non-original ideas, contained in the following text, have been derived from a number of sources, ... It is felt, however, that acknowledgement should be made to Prof. John von Neumann and to Dr. Herman Goldstein for many fruitful discussions ..."
30. ^ Campbell-Kelly, Martin (April 1982). "The Development of Computer Programming in Britain (1945 to 1955)". *IEEE Annals of the History of Computing*. **4** (2): 121–139. doi:10.1109/MAHC.1982.10016. S2CID 14861159.
31. ^ Campbell-Kelly, Martin (1980). "Programming the EDSAC". *IEEE Annals of the History of Computing*. **2** (1): 7–36. doi:10.1109/MAHC.1980.10009.
32. ^ "1985 Computer Pioneer Award 'For assembly language programming' David Wheeler".
33. ^ Wilkes, Maurice Vincent (1949). "The EDSAC – an Electronic Calculating Machine". *Journal of Scientific Instruments*. **26** (12): 385–391. Bibcode:1949JScI...26..385W. doi:10.1088/0950-7671/26/12/301.
34. ^ da Cruz, Frank (2019-05-17). "The IBM 650 Magnetic Drum Calculator". Computing History - A Chronology of Computing. Columbia University. Archived from the original on 2020-02-15. Retrieved 2012-01-17.
35. ^ Pettus, Sam (2008-01-10). "SegaBase Volume 6 - Saturn". Archived from the original on 2008-07-13. Retrieved 2008-07-25.
36. ^ Kauler, Barry (1997-01-09). *Windows Assembly Language and Systems Programming: 16- and 32-Bit Low-Level Programming for the PC and Windows*. CRC Press. ISBN 978-1-48227572-8. Retrieved 2020-03-24. "Always the debate rages about the applicability of assembly language in our modern programming world."
37. ^ Hsieh, Paul (2020-03-24) [2016, 1996]. "Programming Optimization". Archived from the original on 2020-03-24. Retrieved 2020-03-24. "... design changes tend to affect performance more than ... one should not skip straight to assembly language until ..."
38. ^ "TIOBE Index". TIOBE Software. Archived from the original on 2020-03-24. Retrieved 2020-03-24.

39. ^ Rusling, David A. (1999) [1996]. "Chapter 2 Software Basics". *The Linux Kernel*. Archived from the original on 2020-03-24. Retrieved 2012-03-11.
40. ^ [a] [b] Markoff, John Gregory (2005-11-28). "Writing the Fastest Code, by Hand, for Fun: A Human Computer Keeps Speeding Up Chips". *The New York Times*. Seattle, Washington, USA. Archived from the original on 2020-03-23. Retrieved 2010-03-04.
41. ^ "Bit-field-badness". *hardwarebug.org*. 2010-01-30. Archived from the original on 2010-02-05. Retrieved 2010-03-04.
42. ^ "GCC makes a mess". *hardwarebug.org*. 2009-05-13. Archived from the original on 2010-03-16. Retrieved 2010-03-04.
43. ^ Hyde, Randall. "The Great Debate". Archived from the original on 2008-06-16. Retrieved 2008-07-03.
44. ^ "Code sourcery fails again". *hardwarebug.org*. 2010-01-30. Archived from the original on 2010-04-02. Retrieved 2010-03-04.
45. ^ Click, Cliff; Goetz, Brian. "A Crash Course in Modern Hardware". Archived from the original on 2020-03-24. Retrieved 2014-05-01.
46. ^ "68K Programming in Fargo II". Archived from the original on 2008-07-02. Retrieved 2008-07-03.
47. ^ "BLAS Benchmark-August2008". eigen.tuxfamily.org. 2008-08-01. Archived from the original on 2020-03-24. Retrieved 2010-03-04.
48. ^ "x264.git/common/x86/dct-32.asm". git.videolan.org. 2010-09-29. Archived from the original on 2012-03-04. Retrieved 2010-09-29.
49. ^ Bosworth, Edward (2016). "Chapter 1 – Why Study Assembly Language". *www.edwardbosworth.com*. Archived from the original on 2020-03-24. Retrieved 2016-06-01.
50. ^ https://www-01.ibm.com/servers/resourcelink/svc00100.nsf/pages/zOSV2R3sc236852/$file/id
51. ^ Paul, Matthias R. (2001) [1996], "Specification and reference documentation for NECPINW", *NECPINW.CPI - DOS code page switching driver for NEC Pinwriters* (2.08 ed.), FILESPEC.TXT, NECPINW.ASM, EUROFONT.INC from NECPI208.ZIP, archived from the original on 2017-09-10, retrieved 2013-04-22
52. ^ Paul, Matthias R. (2002-05-13). "[fd-dev] mkeyb". *freedos-dev*. Archived from the original on 2018-09-10. Retrieved 2018-09-10.

Further reading

- Bartlett, Jonathan (2004). *Programming from the Ground Up - An introduction to programming using linux assembly language*.

Bartlett Publishing. ISBN 0-9752838-4-7. Archived from the original on 2020-03-24. Retrieved 2020-03-24. [4]

- Britton, Robert (2003). *MIPS Assembly Language Programming*. Prentice Hall. ISBN 0-13-142044-5.
- Calingaert, Peter (1979) [1978-11-05]. Written at University of North Carolina at Chapel Hill. Horowitz, Ellis (ed.). *Assemblers, Compilers, and Program Translation*. Computer software engineering series (1st printing, 1st ed.). Potomac, Maryland, USA: Computer Science Press, Inc. ISBN 0-914894-23-4. ISSN 0888-2088. LCCN 78-21905. Retrieved 2020-03-20. (2+xiv+270+6 pages)
- Duntemann, Jeff (2000). *Assembly Language Step-by-Step*. Wiley. ISBN 0-471-37523-3.
- Kann, Charles W. (2015). "Introduction to MIPS Assembly Language Programming". Archived from the original on 2020-03-24. Retrieved 2020-03-24.
- Kann, Charles W. (2021). "Introduction to Assembly Language Programming: From Soup to Nuts: ARM Edition"
- Norton, Peter; Socha, John (1986). *Peter Norton's Assembly Language Book for the IBM PC*. New York, USA: Brady Books.
- Singer, Michael (1980). *PDP-11. Assembler Language Programming and Machine Organization*. New York, USA: John Wiley & Sons.
- Sweetman, Dominic (1999). *See MIPS Run*. Morgan Kaufmann Publishers. ISBN 1-55860-410-3.
- Waldron, John (1998). *Introduction to RISC Assembly Language Programming*. Addison Wesley. ISBN 0-201-39828-1.
- Yurichev, Dennis (2020-03-04) [2013]. "Understanding Assembly Language (Reverse Engineering for Beginners)" (PDF). Archived (PDF) from the original on 2020-03-24. Retrieved 2020-03-24.
- "ASM Community Book". 2009. Archived from the original on 2013-05-30. Retrieved 2013-05-30. ("An online book full of helpful ASM info, tutorials and code examples" by the ASM Community, archived at the internet archive.)

External links

Assembly language
at Wikipedia's sister projects

- Definitions from Wiktionary
- Media from Wikimedia Commons
- News from Wikinews
- Textbooks from Wikibooks
- Resources from Wikiversity

- Assembly language at Curlie
- Unix Assembly Language Programming
- Linux Assembly
- PPR: Learning Assembly Language
- NASM – The Netwide Assembler (a popular assembly language)
- Assembly Language Programming Examples
- Authoring Windows Applications In Assembly Language
- Assembly Optimization Tips by Mark Larson
- The table for assembly language to machine code

Computer programming portal

By: Wikipedia.org
Edited: 2021-06-18 12:37:12
Source: Wikipedia.org