Assembly language [UNIT-III]

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, often metaprogramming (through macros), certain assemblers are structured or object-oriented
First appeared	1947; 77 years ago

Typing discipline	None
Filename extensions	`.asm`, `.s`, `.inc`, `.wla`, `.SRC` and several others depending on the assembler

In computer programming, **assembly language** (alternatively **assembler language**[1] or **symbolic machine code**),[2][3][4] often referred to simply as **assembly** and commonly abbreviated as **ASM** or **asm**, is any low-level programming language with a very strong correspondence between the instructions in the language and the architecture's machine code instructions.[5] Assembly language usually has one statement per machine instruction (1:1), but constants, comments, assembler directives,[6] symbolic labels of, e.g., memory locations, registers, and macros[7][1] are generally also supported.

The first assembly code in which a language is used to represent machine code instructions is found in Kathleen and Andrew Donald Booth's 1947 work, *Coding for A.R.C.*.[8] Assembly code is converted into executable machine code by a utility program referred to as an *assembler*. The term "assembler" is generally attributed to Wilkes, Wheeler and Gill in their 1951 book *The Preparation of Programs for an Electronic Digital Computer*,[9] who, however, used the term to mean "a program that assembles another program consisting of several sections into a single program".[10] The conversion process is referred to as *assembly*, as in *assembling* the source code. The computational step when an assembler is processing a program is called *assembly time*.

Because assembly depends on the machine code instructions, each assembly language[nb 1] is specific to a particular computer architecture.[11][12][13]

Sometimes there is more than one assembler for the same architecture, and sometimes an assembler is specific to an operating system or to particular operating systems. Most assembly languages do not provide specific syntax for operating system calls, and most assembly languages can be used universally with any operating system,[nb 2] 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, much more complicated tasks than assembling.

In the first decades of computing, it was commonplace for both systems programming and application programming to take place entirely in assembly language. While still irreplaceable for some purposes, the majority of programming is now conducted in higher-level interpreted and compiled languages. In "No Silver Bullet", Fred Brooks summarised the effects of the switch away from assembly

language programming: "Surely the most powerful stroke for software productivity, reliability, and simplicity has been the progressive use of high-level languages for programming. Most observers credit that development with at least a factor of five in productivity, and with concomitant gains in reliability, simplicity, and comprehensibility."[14]

Today, it is typical to use small amounts of assembly language code within larger systems implemented in a higher-level language, for performance reasons or to interact directly with hardware in ways unsupported by the higher-level language. For instance, just under 2% of version 4.9 of the Linux kernel source code is written in assembly; more than 97% is written in C.[15]

Assembly language syntax

Assembly language uses a mnemonic to represent, e.g., each low-level machine instruction or opcode, each directive, typically also each architectural register, flag, etc. Some of the mnemonics may be built-in and some user-defined. 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.

Some are column oriented, with specific fields in specific columns; this was very common for machines using punched cards in the 1950s and early 1960s. Some assemblers have free-form syntax, with fields separated by delimiters, e.g., punctuation, white space. Some assemblers are hybrid, with, e.g., labels, in a specific column and other fields separated by delimiters; this became more common than column-oriented syntax in the 1960s.

Terminology

- A **macro assembler** is an assembler that 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.
  - **Open code** refers to any assembler input outside of a macro definition.
- 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 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",[16] or that accepts an assembler source file along with such a description and assembles the source file in accordance with that description. "Meta-Symbol" assemblers for the SDS 9 Series and SDS Sigma series of computers are meta-assemblers.[17] Sperry Univac also provided a Meta-Assembler for the UNIVAC 1100/2200 series.[18]
- **inline assembler** (or **embedded assembler**) is assembler code contained within a high-level language program.[19] 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.[20] 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,[20] 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.[21]

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** process the source code once. For symbols used before they are defined, the assembler will emit "errata" after the eventual definition, telling the linker or the loader to patch the locations where the as yet undefined symbols had been 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.[22]

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[23]

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 declarative operations, directives, pseudo-instructions, pseudo-operations 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.[24] 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.[24]

```
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 3] 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 lack of ambiguity 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.[24]

```
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.[25]

```
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
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.[24]

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 4] 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*.[26]

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`.[27] 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

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".[20] 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.[28]

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 5] 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.[29]

Macro assemblers typically have directives to, e.g., define macros, define variables, set variables to the result of an arithmetic, logical or string expression, iterate, conditionally generate code. Some of those directives may be restricted to use within a macro definition, e.g., **MEXIT** in HLASM, while others may be permitted within open code (outside macro definitions), e.g., **AIF** and **COPY** in HLASM.

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.[30] 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: