# MACHINE LANGUAGE

**David R. Kaeli**
*Department of Electrical and Computer Engineering, Northeastern University, USA*

**Keywords:** microperations, data path, assembly code, binary format, microcontroller, microcode, macroinstruction,

**Contents**





**Summary**

Machine Language is the representation of a program that either executes directly on the central processing unit or is represented in an assembly language that is later compiled down to binary code. This article discusses machine language, its different forms, and the microarchitectural data path that supports the execution of the machine language.

## 1. Introduction

Computer applications are written at different levels of abstraction, including high-level languages, assembly code, and binary code. High-level languages such as C, C++ and Java, provide an abstract view of a computing system. These languages depend upon either a compiler or an interpreter to carry out the desired operations as specified in the program. High-level languages are eventually transformed into binary code, which can be executed directly on the underlying hardware. The use of high-level languages has increased the productivity of the programmer, and has provided software developer with an efficient abstraction from the underlying computer system.

Assembly code is a language that provides a symbolic (readable) intermediate version of the program. Programmers can implement their applications in assembly code. This

model is commonly used in the embedded computing field, where a small, compact program controls the underlying hardware.

Binary code is a machine-readable form of the assembly code. While binary instrumentation systems are able to write binary code, typically programmers develop their applications at a higher level. There exist standard formats for binary code, including image formats, data layout, and the target instruction set of the computer.

Both binary and assembly code are forms of machine language. This article will provide an overview of a typical assembly language, as well as a description a typical binary code format. Finally, we will discuss how the machine language is executed on the hardware, and describe some of the issues associated with machine code execution.

## 2. Assembly Language

An assembly language (sometimes referred to assembler or assembly) provides the program almost complete control with the actual realization of their program. The benefits of writing in assembly code are that the programmer has a lot more control over the contents of the final binary code. The assembler (the program which transforms assembly code to binary code) can still apply some optimizations, though the executed binary code will closely resemble the assembly code.

An assembly language provides the programmer with a set of well-defined primitives. These primitives are expressed on a single line in the program. Each lines contains three fields:

- A label that is used to identify unique points in the program,
- An instruction that specifies the operation desired (including operands), and
- A comment, indicating the purpose or ramifications of executing this instruction.

While fields 1 and 3 are optional, every line in an assembly language contains an instruction field.

High-level languages will generate multiple assembly instructions per source code line, while a single binary level instruction. Presently, most embedded software is developed in assembly since the program has a lot more control over the final binary.

The reason why many programmers also use high-level languages is a tradeoff between high performance and productivity.
The reason why programmers do not write their programs at the binary level is that they can still obtain the control over the final binary by writing in assembly code, though they do not have to worry about the actual bit values in each instruction, and can instead work with symbols (e.g., Loop1) and instruction operation abbreviations (e.g., MOV, ADD and CMP, to denote move, addition and compare operations, respectively).

Note that the above assembly program is written in a format that is compatible with a particular assembler. Each assembler will demand a particular specification of the assembly code format. The code can be assembled and run on a number of different

versions of the Intel 80X86 microprocessor family (e.g., the 80486, PentiumI, PentiumII, PentiumIII, PentiumIV, etc.).

```
Label       Operation and Operands     Comment


Loop1:      MOV  EAX,(100)             ; load first operand
            MOV  EBX,(200)             ; load second operand
            ADD  EDX,EAX,EBX           ; add the operands
            MOV  (100),EDX             ; store result in location of first operand
            CMP  EDX,0x100             ; check loop
            BNE  Loop1          ; branch to top of the loop if  not equal
```

Figure 1 shows an example of an assembly code format.

### 2.1 Assembler Directives

To raise the abstraction level in assembly-level programming, assembler directives (i.e., macros) can be used. These directives are used tell the assembler to perform some task. Some of these tasks include allocating storage, aligning code or data on a memory boundary, or performing conditional execution of portions of a program.

```
A:      DW    100
B:      DW    200


IF A EQU 100
        C:    DW    300
ELSE
        C:    DW    400
ENDIF
```

Figure 2. An example of assembly directives for the Intel Pentium

In Figure 2 we provide an example of assembly directives. The first two lines tell the assembler to allocate enough storage for two doublewords (a doubleword on the Intel Pentium is 32 bits), give these locations the labels of A and B, and initialize their values to 100 and 200, respectively. Most assembly languages provide the programmer with a number of shortcuts that reduce the

## 3. Binary Code

Once programs have been compiled, they will generate object files. A single program many comprise many object files. These files are linked together to form an executable binary. There are a number of formats for executable files. Some of these include

ELF
COEFF
Java Class File

### 3.1 ELF

The executable and linking format (ELF) was originally developed by Unix System Laboratories and is now a very popular executable file format. ELF is the default binary format on operating systems such as Linux and Solaris. ELF executable files contain executable code, sometimes referred to as text and data sections. Each executable image contains tables, which describe how the image should be mapped into virtual memory. Some of the capabilities of ELF include: dynamic linking, dynamic loading, imposing runtime control on a program, and an improved method for creating shared libraries. The ELF representation of control data in an object file is platform independent, an additional improvement over previous binary formats. The ELF representation permits object files to be identified, parsed, and interpreted similarly, making the ELF object files compatible across multiple platforms and architectures of different size.

The three main types of ELF files are executable, relocatable, and shared object files. These file types hold the code, data, and information about the program that the operating system and/or link editor needs to read. The three types of files are summarized as follows:

- An executable file supplies information necessary for the operating system to create a process image suitable for executing the code and accessing the data contained within the file.
- A relocatable file describes how it should be linked with other object files to create an executable file or shared library.
- A shared object file contains information needed in both static and dynamic linking.

### 3.2 COFF

The Common Object File Format (COFF) is the format used by Microsoft for its object files. A full specification of COFF can be found at the Microsoft Developer Network.

Each COFF executable will contain the following structures:

- File Header – contains an overview of the file and controls the layout of other sections
- Optional Header – used to store the instruction pointer address for executable files
- Section Header - maintains the location and size information about code and data sections
- Section Data - contains code and data for the program
- Relocation Directives – contains fix-up information needed when relocating a section
- Line Numbers – holds the address of each line number in the code/data section
- Symbol Table – contains symbol information relating locations in the source code to code in the executable
- String Table – stores symbol names

Some of the main features of COFF include:

- applications can add system-dependent information to the object file without causing access utilities to become obsolete
- space is provided for symbolic information used by debuggers and other applications, and
- programmers can modify the way the object file is constructed by providing directives at compile time

-
-
-



**Bibliography**

1. (1985). *IEEE Standard for Microprocessor Assembly Language. (IEEE Std. 694-1985)*, New York, NY, Institute of Electrical and Electronic Engineers [IEEE specification of microprocessor assembly language format]

2. Irvine K.R. (1999). *Assembly Language for Intel-Based Computers*, 3rd edition, Prentice Hall, Upper Saddle River, NJ [A reference guide for assembly code targeting the 80X86 family of microprocessors.]

3. Microsoft Developers Network, http://msdn.microsoft.com [Extensive information on the COFF binary format.]

4. (1990). UNIX Software Operations, UNIX System V Release 4 Programmers Guide: Ansi C and Programming Support Tools, *STREAMS*, Prentice Hall [Extensive information on the ELF binary format.]

5. Venners B. (1996). The Java Class File Lifestyle, *Artima Software Company* [Extensive information on the Java Class File format.]

6. Wilkes M.V. (1951). The Best Way to Design an Automatic Calculating Machine, *Report of Manchester University Computer Inaugural Conference*, 16-18, July [The seminal work on microprogramming.]

7. Mano M.M. and Kime C.R. (2000). *Logic and Computer Design Fundamentals*, 2nd edition, Prentice-Hall, Inc., Upper Saddle River, NJ [A standard reference for machine organization and design.]

8. Tannenbaum A.S. (1990). Structured Systems Architecture, 3rd edition, Prentice Hall, Upper Saddle River, NJ [A standard reference for machine organization and design.]

9. Stallings W. (2000). *Computer Organization and Architecture*, 5th edition, Prentice Hall, Upper Saddle River, NJ [A higher level text on computer architecture.]

10. Hennessy J.L. and Patterson D.A. (1996). *Computer Architecture: A Quantitative Approach*, 2nd edition, Morgan Kaufman, San Francisco, CA [An advanced text on computer architecture.]

11. Patterson D.A. and Hennessy J.L. (1994). *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufman, San Francisco, CA [An intermediate level text on computer architecture.]