# 实验 2：汇编语言及程序设计

## 背景知识

## 1. Assembly Language

An **assembly language** is a low-level programming language for computer, microprocessors, microcontrollers, and other programmable devices. It implements a symbolic representation of the machine codes and other constants needed to program a given CPU architecture. This representation is usually defined by the hardware manufacturer, and is based on mnemonics that symbolize processing steps (instructions), processor registers, memory locations, and other language features. An assembly language is thus specific to certain physical (or virtual) computer architecture. This means that each computer architecture and processor architecture usually has its own machine language.

A utility program called an *assembler* is used to translate assembly language statements into the target computer's machine code. The assembler performs a more or less isomorphic translation (a one-to-one mapping) from mnemonic statements into machine instructions and data. This is in contrast with high-level language, in which a single statement generally results in many machine instructions.

## 2. Key concepts

### Assembler

Typically a modern **assembler** creates *object code* by translating assembly instructions mnemonics into *opcodes*, and by resolving *symbolic names* for memory locations and other entities. The use of symbolic reference 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*.

### Number of passes

There are two types of assemblers based on how many passes through the source are needed to produce the executable program.

- One-pass assemblers go through the source code once and assume that all symbols will be defined before any instruction that references them.
- Two-pass assemblers create a table with all symbols and their values in the first pass, and then

use the table in a second pass to generate code. The assembler must at least be able to determine the length of each instruction on the first pass so that the addresses of symbols can be calculated.

The advantage of a one-pass assembler is speed, which is not as important as it once was with advances in computer speed and abilities. The advantage of the two-pass assembler is that symbols can be defined anywhere in program source code. This lets programs be defined in more logical and meaningful ways, making two-pass assembler programs easier to read and maintain.

**Assembly language program**

A program written in assembly language consists of a series of mnemonic statements and meta-statements (known variously as directives, pseudo-instructions and pseudo-ops), comments and data. These are translated by an assembler to a stream of executable instructions that can be loaded into memory and executed. Assemblers can also be used to produce blocks of data, from formatted and commented source code, to be used by other code.

# 3. 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 sections
- Assembly directives

**Opcode mnemonics and extended mnemonics**

Instructions (statements) in assembly language are generally very simple, unlike those in high-level language. 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 (typically one byte values, coded in the instruction itself), registers specified in the instruction, 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.

*Extended mnemonics* are often used to support specialized uses of instructions, often for purpose 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 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*. These are sometimes known as *pseudo-opcodes*.

**Data sections**

There are instructions used to define data elements to hold data and variables. They define the type 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 instructions that are executed by an assembler at assembly time, not by a CPU at run time. They can make the assembly of the program dependent on parameters input by a programmer, so that one program can be assembled different ways, perhaps for different applications. They also can be used to manipulate presentation of a program to make it easier to read and maintain. For example, directives would be used to reserve storage areas and optionally their initial contents. The names of directives often start with a dot to distinguish them from machine instructions.

Symbolic assemblers let programmers associate arbitrary names (*labels* or *symbols*) with memory locations. Usually, every constant and variable is given a name so instructions can reference those locations by name, thus promoting self-documenting code. In executable code, the name of each subroutine is associated with its entry point, so any calls to a subroutine can use its name. Inside subroutines, GOTO destinations are given labels. Some assemblers support *local symbols* which are lexically distinct from normal symbols.

Assembly languages, like most other computer languages, allow comments to be assembly source code that are ignored by the assembler. Good use of comments is even more important with assembly code than with higher-level languages, as the meaning and purpose of a sequence of instructions is harder to decipher from the code itself.

**Macros**

Many assemblers support *predefined macros*, and others support *programmer-defined* (and repeatedly re-definable) macros involving sequences of text lines in which variables and constants are embedded. This sequence of text lines may include opcodes or directives. Once a macro has been defined its name may be used in place of a mnemonic. When the assembler processes such a statement, it replace 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).

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 language. 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.

Many assemblers have built-in (or *predefined*) macros for system calls and other special code sequences, such as the generation and storage of data realize through advanced bitwise and boolean operations used in gaming, software security, data management, and cryptography.

(Excerpted from http://en.wikipedia.org/wiki/Assembly_language)

# 实验目的

- 了解汇编语言(assembly language)与机器指令（指令集体系结构）之间的关系
- 掌握汇编程序(assembler)的基本内容和实现原理
- 掌握汇编语言程序设计

# 实验内容

1. 定义 NK-CPU 汇编语言的基本内容
2. 实现 NK-CPU 的汇编程序
3. 使用 NK-CPU 汇编语言编写对 20 个随机数的排序程序

# 要求

NK-CPU 的汇编程序需要使用 C 语言或 C++语言实现。