目录

第 1 章 绪论

本章主要介绍汇编语言和机器语言的概念，剖析汇编语言、机器语言和高级语言之间的关系，阐述学习汇编语言的重要性和学习方法。本章给出了一个汇编语言源程序的例子，通过该例介绍汇编语言源程序的基本结构和格式。通过本章的学习，应深刻理解计算机世界是 0-1 世界的本质，计算机中所有的信息都是由 0 和 1 组成。探索现实世界中的信息如何编码成计算机世界中 0-1 串以及 0-1 串又如何解码对应到现实世界的奥妙。

1．1 什么是汇编语言？

1.1.1 机器语言

学习过 C 语言程序设计的人都知道，在编写 C 语言程序之后，需要对这个程序进行编译和链接，生成可执行程序 exe 文件。之后，可以运行该程序，也可以将其拷贝或传送到其他位置。那么，执行程序是什么样子的呢？换句话说，执行程序里面存放的是什么呢？

设有如下 C 语言程序 c_example.c，生成的可执行文件 c_example.exe。

```
#include <stdio.h>
int main(int argc, char* argv[])
{
    int   x, y, z;
    x = 10;
    y = 20;
    z = 3 * x + 6 * y+ 4*8;
    printf("3*%d+6*%d+4*8=%d\n",x,y,z);
    return 0;
}
```

如图 1.1 所示，用二进制编辑器打开可执行文件 c_example.exe，就会发现它都是一些 16 进制数字串。

```
4D 5A 90 00 03 00 00 00  04 00 00 00 FF FF 00 00
B8 00 00 00 00 00 00 00  40 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00  00 00 00 00 E8 00 00 00
0E 1F BA 0E 00 B4 09 CD  21 B8 01 4C CD 21 54 68
69 73 20 70 72 6F 67 72  61 6D 20 63 61 6E 6E 6F
74 20 62 65 20 72 75 6E  20 69 6E 20 44 4F 53 20
6D 6F 64 65 2E 0D 0D 0A  24 00 00 00 00 00 00 00
```

图 1.1 机器语言程序片段示例

操作提示： 在 Visual Studio 2019 中，单击“文件”->“打开文件”，在打开文件对话框中选择要打开的文件，并在“打开方式”中选择“二进制编辑器”，即可显示图 1.1 所示内容。

图 1.1 给出的是一个机器语言程序片段。机器语言程序是由机器指令组成的。机器指令

（Machine Instruction）也常常被称为硬指令，它是面向机器的，不同的 CPU（Central Processing Unit，中央处理器）都规定了自己所特有的、一定数量的基本指令，这批指令的全体即为计算机的指令系统。这种机器指令的集合就是机器语言（Machine Language）。用机器语言编写的程序称为机器语言程序。

在计算机开始出现的年代，将由 0 和 1 组成的程序代码打在纸带或卡片上，1 打孔，0 不打孔，再通过纸带机或卡片机将程序输入到计算机中运行。在那个时候，程序员编写由 0 和 1 组成的程序，然后由专门人员负责在纸带打孔和填补误穿的孔。如图 1.2，展现出来是编码为 “83H　06H”的纸带示意图。

图 1.2　表示 83H　06H 的纸带

机器指令由一个字节或者多个字节组成，其中包括操作码字段、一个或多个有关操作数地址的字段、操作数（立即数）等等。操作码和地址码均是由 0 和 1 组成的二进制代码。操作码指出了操作的种类，如加、减、乘、除、传送、移位、转移等；地址码指出了参与运算的操作数和运算结果存放的位置。除此操作码和地址码之外，还有指令中一些信息的编码。在 4.9 节中详细介绍了 Intel CPU 机器指令的编码规则。

提示：(1) 机器指令是 0-1 串，但并非任意一个 0-1 串都可以解析成机器指令；

(2) 不同的 CPU 的机器指令是不同的；

(3) 能够被 CPU 解释执行的只有机器指令；

(4) 可执行文件是有特定的结构的，除了程序的机器指令外，还有许多其他信息。在附录一中介绍了可执行文件的结构。

1.1.2 汇编语言

虽然在 CPU 上能够被解释执行的只有机器语言程序，但是用机器指令编写程序相当麻烦，写出的程序也难以阅读和调试。为了克服这些缺点，人们就想出了用助记符表示机器指令的操作码；用变量代替操作数的存放地址；在指令前冠以标号用来代表该指令的存放地址等。这种用符号书写的、其主要操作与机器指令基本上一一对应的、并遵循一定语法规则的计算机语言就是汇编语言（Assembly Language）。用汇编语言编写的程序称为汇编源程序。由此可见，汇编语言也是面向机器的语言。

为了让读者对汇编语言有一个感性的认识，我们首先以反汇编的形式显示可执行程序。所谓反汇编就是将由 0-1 组成的机器码翻译成用符号表达的形式。如图 1.3 所示，给出了 1.1.1 节中 C 语言程序（c_example.exe）的反汇编窗口中显示内容的片段。从图中可以看到各条指令的机器码以及对应的汇编语言指令。具体的操作方法见第 19 章上机操作。

```
x = 10;
00251828  C7 45 F8 0A 00 00 00 mov     dword ptr [ebp-8],0Ah
y = 20;
0025182F  C7 45 EC 14 00 00 00 mov     dword ptr [ebp-14h],14h
z = 3 * x + 6 * y+ 4*8;
00251836  6B 45 F8 03           imul    eax,dword ptr [ebp-8],3
0025183A  6B 4D EC 06           imul    ecx,dword ptr [ebp-14h],6
0025183E  8D 54 08 20           lea     edx,[eax+ecx+20h]
00251842  89 55 E0              mov     dword ptr [ebp-20h],edx
```

图 1.3　执行程序反汇编后的片段

从图 1.3 中可以看到：语句“x=10;”对应的机器指令“C7 45 F8 0A 00 00 00”，对应的汇编语言语句是：mov dword ptr [ebp-8],0Ah。该语句完成的功能是将 10（16 进制为 0Ah）送到地址为[ebp-8]的双字存储单元中,其中，mov 为数据传送指令的助记符，代表了机器指令中的操作码；[ebp-8]表示在当前堆栈段中的一个单元，它是变量 x 的地址；“dword ptr”说明了这个目的操作数是 32 位二进制数，而源操作数是 0Ah，用 32 位来表达等同于 0000000AH。

在上面的机器指令“C7 45 F8 0A 00 00 00”中，“0A 00 00 00”，表示了一个数 0000000AH，即十进制数值 10，它是一个双字的数据，数据的低字节放在地址小的单元中，数据的高字节放在地址大的单元中。将 F8H 当成一个字节的有符号数的补码表示，它对应的是 -8。在前面还有 C7H 45H，表示要进行“数据传送”操作，还指明了以何种方式取得源操作数和目的操作数。较详细的机器指令编码规则参见 4.9 节。

在机器指令的左边，有数字 00251828，这是指令在内存中存放的起始地址。在程序调试时，打开内存显示窗口，输入内存的起始地址，可以看到以该地址为起始地址的一片存储单元中的内容，如图 1.4 所示。

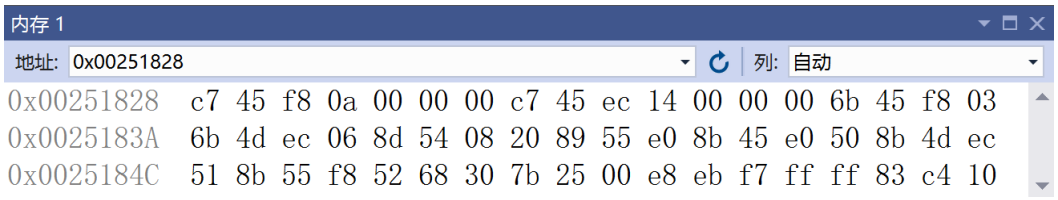

图 1.4 内存显示窗口

在内存窗口，在“C7 45 F8 0A 00 00 00”之后是“C7 45 EC 14 00 00 00”，它是“y=20;”的机器码。“6B 45 F8 03”是“imul eax,dword ptr [ebp-8],3”的机器码。

由语句“x=10;” 的地址是 00251828、该指令占 7 个字节、下一条指令紧接在前一条指令之后可知，语句“y=20;”的地址是 0025182F（00251828H+7=0025182FH）。

对于 z = 3 * x + 6 * y+ 4*8; 它被翻译成了 4 条指令：

```
imul   eax, dword ptr [ebp-8],   3     ;   (x)*3 -> eax
imul   ecx,dword ptr [ebp-14h], 6      ;   (y)*6 ->ecx
lea    edx, [eax+ecx+20h]              ;   (eax) + (ecx) + 20h ->edx
mov    dword ptr [ebp-20h], edx        ;   (edx) -> z
```

直观上看，实现“z = 3 * x + 6 * y+ 4*8;”要分成几个步骤来完成，首先是执行 3*x，结果放在寄存器 eax 中；再执行 6*y，结果放入另一个寄存器 ecx 中；4*8 是一个常量表达式，在编译时，计算了该表达式的值为 32，即 20H，之后执行三个数相加，结果送到寄存器 edx 中，最后将 edx 中的内容送入变量 z 中。

虽然计算机 CPU 能识别和执行的是机器语言程序，但是用机器语言编写程序是非常困难的。汇编语言是为了方便用户而设计的一种符号语言，因此，用它编写出的源程序并不能直接被计算机识别，必须要将它翻译成由机器指令组成的程序后，计算机才能识别并执行。这种由源程序经过翻译转换，生成的机器语言程序也称为目标程序。目标程序中的二进制代码(即机器指令)称为目标代码。这个翻译工作一般都由计算机自己去完成，但人们事先必须将翻译方法编写成一个语言加工程序作为系统软件的一部分，在需要时让计算机执行这个程序才可完成对某一汇编源程序的翻译工作。这种把汇编源程序翻译成目标程序的语言加工程序称为汇编程序（即编译器）。汇编程序进行翻译的过程称为汇编（编译）。

在这里，汇编程序相当于一个翻译器，它加工的对象是汇编源程序，而加工的结果是目标程序。它们三者之间的关系如图 1.5 所示。当然，在 Visual Studio 中集成了编译器，可直接在该开发平台下编写汇编源程序，然后编译、链接、调试和执行。

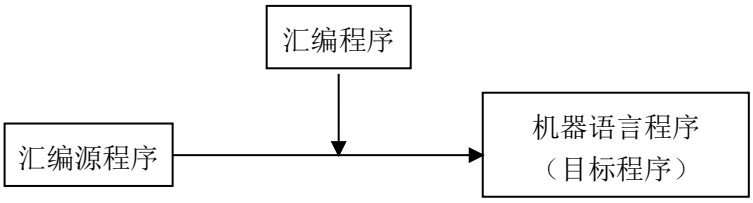

图 1.5　汇编程序与汇编源程序、目标程序的关系

为了能让汇编程序正确地完成翻译工作，必须要告诉汇编程序，源程序应从什么位置开始安放，汇编到什么位置结束，数据放在什么位置，数据的类型是什么，留多少内存单元作临时存储区等。这就要求源程序中应该有一套告诉汇编程序如何进行汇编工作的命令，这种命令称伪指令(或称汇编控制命令)。由此可见，指令助记符、语句标号、数据变量、伪指令及它们的使用规则构成了整个汇编语言的内容。

由于汇编语句基本上与机器指令对应，因而它的编写也是相当麻烦的。为了简化程序的编写，提高编程效率，Visual Studio 提供的编译器有很多新特性，支持很多接近于 C 语言的伪指令，使得用汇编语言写出的程序有点像 C 语言程序。当然，对于初学者首先要掌握机器指令，在使用条件流控制、函数调用等类似于高级语言语句的伪指令时，要清楚这些伪指令的编译结果，即它们与机器语言的对应的关系。

本教材中所介绍的是 Intel X86 汇编语言，适用对象是 Intel 公司的 X86 系列 CPU。虽然 Intel 公司的 CPU 的型号很多，新产品不断涌现，但它们的机器语言和汇编语言保持了很好的兼容性，新型的 CPU 一般都是在原有的指令系统基础上增加一些新的机器指令。

1．2 为什么学习汇编语言？

与机器语言相比，汇编语言易于理解和记忆，所编写的源程序也容易阅读和调试，所占用的存储空间、执行速度与机器语言相仿。虽然与机器语言相比，使用汇编语言编写程序要简单很多，但是与高级语言相比，编写程序还是相当的麻烦。简单的任务也需要很多汇编语言语句，在进行程序设计时必须面面俱到，需要考虑到各种可能的问题，以合理调配和使用各种软、硬件资源。这样，就不可避免地加重了程序员的负担。在程序调试时，一旦程序的运行出了问题，就很难发现。

例如，对于一条 C 语句，“z = 3 * x + 6 * y+ 4*8;”，对应的汇编语句是：

```
imul    eax,dword ptr [x],3    ; 将 x 单元中的内容与 3 相乘，结果放在 eax 中
imul    ecx,dword ptr [y],6    ; 将 y 单元中的内容与 6 相乘，结果放在 ecx 中
lea     edx,[eax+ecx+20h]    ; 将 eax、ecx、20H（即 4*8）的内容相加，结果送入 edx 中
mov     dword ptr [z],edx     ; 将 edx 中的内容送入 z 单元中
```

提示：上述语句与图 1.3 所示的反汇编语句略有不同，但是机器码是完全相同的，只是在反汇编窗口的“查看选项”中勾选了“显示符号名”，即用符号名 x 代替了地址表达式[ebp-8]。采用符号名更直观一些，但其本质是一个存储单元地址的符号表示。

从这个简单的例子可以看到用汇编语言写程序是很繁琐的。

纵观计算机程序设计语言的发展历史，可以看到计算机编程语言是从机器语言、汇编语言向高级语言发展的。其目标就是要让程序员编写程序越来越简单。高级语言接近于自然语言，易学、易记、便于阅读、容易掌握、使用方便、通用性强，且不依赖于具体的计算机。在使用高级语言编写程序后，只需要配备相应的编译器，将其翻译成机器语言程序即可。从 C 语言，到面向对象的 C++，再到 Java 和 Python 语言以及各种开发平台的出现，都反映了将程序员从繁重的一般性的脑力劳动中解放出来的进步，反映了降低编写程序这一部分的工作量，同时提高程序开发效率，使他们能够更多的集中精力分析需求、设计和研究高效算法

的发展趋势。

毋庸置疑，在现在的 IT 界很少使用汇编语言编写程序，那么我们为什么要学习汇编语言呢？对于计算机科学与技术专业的学生而言，学习汇编程序设计语言的必要性或者价值主要体现在以下几个方面。

（1）逆向工程、解密程序、病毒和木马分析和防治的唯一选择

当进行程序解密、病毒和木马分析等工作的时候，我们并没有源程序，而只有可执行程序。此时，需要对可执行程序进行反汇编得到汇编语言程序，然后阅读和分析。

（2）理解高级语言的最好途径

虽然现在有许多的高级程序设计语言可以用来编写程序，但是编程者并不一定真正理解程序中语句的执行过程，这有碍于牢固掌握程序设计的基本知识，有碍于灵活应用编程语言。对于一个 C 语言程序而言，地址类型转换和数据类型转换的含义是什么？为什么调用一个函数后能够返回到调用语句的下一行？函数之间是如何传递参数和返回结果的？为什么局部变量的作用域只在函数内部？递归程序如何理解？数组越界访问是怎么回事？指针是如何指向相应对象的？UNION 结构中各成员是什么关系？在 C++程序设计中，对象构造、对象析构、继承、多态、成员对象的引用等等更复杂的内容是如何实现的？在编程者深刻把握语句的执行原理后，编写程序时就可以少犯错误，写出执行效率高且形式优美的程序。掌握这些学科的基础知识，可以帮助程序员提升编程水平。

通过对执行程序的反汇编，或者分析在编译时生成汇编语言程序，可以清楚的看到每一语句对应的一个执行系列，可以清楚地看到变量的空间分配方法、数据结构中各组成部分的空间关系、数据的传递方法、程序执行流程的转移方法，从本质上理解机器的行为。

（3）学习后续专业课程的基础

汇编语言程序可以看成高级语言源程序编译的结果。因此，对于编译原理学习而言，通过汇编语言的学习，可以了解编译程序生成的目标代码。对于操作系统而言，生成的程序由操作系统进行调度执行和管理，从汇编语言程序中可以了解一个源程序如何调用操作系统提供的接口。在学习计算机组成原理的时候，需要了解有关的硬件设备，例如运算器、存储器是如何完成指令加工的，汇编语言提供了要在这些设备上处理的对象。汇编语言也是接口技术和嵌入式系统等的先修课程。

（4）为深入地理解计算机硬件、操作系统、应用程序之间的交互工作奠定基础

汇编语言操作直接面向硬件，指令操作更直接，通过一条一条直接控制计算机的指令，清晰地看到计算机的工作，理解计算机的内部工作方式，从而对计算机硬件和应用程序之间的联系和交互形成一个清晰的认识，形成一个软、硬兼备的编程知识体系。例如，CPU、内存和硬盘等硬件设备如何协调地工作在一起，数据从哪里转移到哪里，在哪里运算和存储等。在深入理解工作原理之后，才能够写出运行效率高的程序。

举一个简单例子，要求编写一个程序，求一个二维数组中所有元素的和。显然，可以用双重循环语句来实现，既可以用行序优先（即第一行元素相加后再加第二行的元素），也可以用列序优先。虽然从算法复杂度的角度来看，每个元素都只被访问了一次，算法复杂度是一样的。但是如果数组较大，监测这两个程序段的运行速度，就会发现用行序优先快于用列序优先的方法。从内存是随机存取存储器（不同于磁带等顺序存取设备）的角度来看，不论给定的是什么存储单元的地址，访问速度是一样的，就无法解释这一现象。但如果更深入的知道计算机的工作方式，了解 CPU 利用了其所带的高速缓冲存储器（Cache）所起的作用，了解数据读取并不是一次只从内存读一个单元，以及二维数组数据在一维内存中的存放方式，了解 Cache 命中率、Cache 抖动等知识，就很容易解答为什么会出现上述现象，从而编写执行效率更高的程序。

在研究国产的 CPU 芯片、国产的操作系统、数据库管理系统、编译系统、国产的软件

开发平台等征程中，都离不开这些计算机核心技术。

（5）特定场合下编写程序的必然选择

在使用高级语言编写程序后，需要编译生成机器语言程序。虽然编译器优化在生成执行时间更短的机器代码上取得了长足的进步，仍然存在一些不能充分利用机器性能的缺陷。在对软件的执行时间或存储容量要求较高的场合，如系统程序的关键核心等，需要软件开发人员利用汇编语言优点进行编程，提高系统的性能。究其原因，汇编语言保持了机器语言的优点，具有直接和简捷的特点，可有效地访问、控制计算机的各种硬件设备，如磁盘、存储器、CPU、I/O 端口等，且占用内存少，执行速度快，是高效的程序设计语言。

当然，整个程序采用汇编语言来编写并不是一个明智的选择或者是不恰当的选择。在本书中，给出了在 C 语言程序中调用汇编语言编写的函数的例子，也给出了在汇编语言源程序中调用 C 语言函数的例子，以及 C 和汇编混合编程的例子。

此外，在软件与硬件关系紧密、软件要直接控制硬件、以及没有合适高级语言的场合，如设备驱动程序，需要直接使用汇编语言编程。

综上所述，学习汇编语言是非常必要的。通过汇编语言的学习，了解计算机内部工作的一些原理，理解高级语言语句的运行过程和对数据的处理方式，为编写高水平的程序打下基础，也为后继课程的学习做好准备。

1．3 如何学习汇编语言？

学好一门课程是有一些规律的。本书力争将这些规律具体化，使得读者能对各个知识点有更透彻的理解。

（1）态度决定一切，兴趣是最好的老师

（2）带着问题来学

没有问题的安逸，如同没有引爆的定时炸弹一样危险。“是什么？为什么？有没有其他方式，哪种方式更好？”等等，都是经常需要问自己的问题。通过猜想或搜寻这些问题的答案，探索其中的奥秘，可以更好的更全面的掌握所学知识。

（3）应用建构主义学习理论

要把不同的内容关联起来，只有知识相互关联起来，形成了网络，才能记得清楚，记得牢固。建构主义学习理论，就是要有从已有的知识产生未知的知识。在学习汇编语言的时候，完全可以和学习过的高级语言程序设计关联起来，建立起一种对应关系，比较它们之间的共同点，寻找它们之间的差异。一方面，通过阅读 C 语言程序编译生成的汇编语言程序，可帮助理解汇编语句的功能，帮助熟悉汇编语言语句的表达方式，仿照编译器生成的汇编语言程序和 C 语言程序的指令流程，有助于编写自己的汇编语言源程序。另一方面，以反汇编为手段，可以更好的学习计算机工作原理、理解计算机世界中信息存储方式、加工过程，帮助理解和分析 C 语言程序设计、数据结构中的一些复杂问题。

著名计算机科学家、1984 年图灵奖得主、Pascal 语言之父 N.Wirth 提出“程序=算法+数据结构”。不论采用何种语言编写程序，它们的本质都是一样的，都离不开算法和数据结构。算法是软件的灵魂，好的算法会给软件带来的往往都是质的变化。算法是处理解决问题的思路及办法，程序语言是按照一定语法把算法表达出来。在学习汇编语言时，完全可以利用 C 语言学习中掌握的算法，即利用所学知识帮助新知识的学习。

数据结构是计算机学科中的重要课程，是程序设计中的关键组成部分，是计算机存储、组织数据的方式。数据结构是指相互之间存在一种或多种特定关系的数据元素的集合。从机器语言的角度看，每一条机器指令都是对数据进行处理，其核心是需要在指令中给出数据的地址。数据结构中的各个数据元素都被编译器转换成有一定关系的存储单元地址。通过汇编

语言的学习，通过对高级语言程序编译后生成的机器语言程序的反汇编，就可以看到元素之间的地址关系，从而更好的理解数据元素之间的关系，更好的理解数据结构。这样有助于牢固掌握数据结构知识，提高灵活应用数据结构的能力。

（4）理解之后的记忆

只有深刻理解了的知识才能牢固地记住它。所谓理解是指当提到某一知识时，头脑中就能够想到跟它有关的事实，知道它的应用或意义，了解它跟有关知识的联系。汇编语言中有较多的语法规则，要记住这些规则，应该先理解制定这些规则的原因，即为什么要制定这些规则。在理解规则存在的合理性后，才能较好的遵循这些规则。例如，在理解计算机的工作原理后，知道 CPU 当前要执行的指令的位置是由 EIP 指示的，那么就会认识到程序跳转的本质是要改变 EIP，这就需要在指令语句中给出转移的目的地址或者以约定的某种特定方式告诉 CPU 如何得到转移的目的地址。在函数执行完成后，要回到调用之处继续执行，又是要改变 EIP，从哪儿取值给 EIP 呢？在调用函数时，给出的函数名就对应一个地址。而一个函数可以被多次在程序的不同位置被调用，不同位置的调用，返回的目的位置都是不同的，不可能在函数中固定写死返回的地址。由此就可以记住，调用函数（子程序）时，要保存断点地址，即在执行函数调用时，CPU 就要将函数执行完后的返回地址存放在某个位置（堆栈中）。因此也不难记住，执行返回时就是从堆栈栈顶取出返回地址送给 EIP。

（5）把书由厚读薄

虽然汇编语言中的指令较多，但是它们的语法、语义也是有一定的共同规律的。对这些指令进行概括和分类，掌握它们的共同规律就可以将书读薄。

例如，在整个汇编语言之中，处于核心地位的是地址，包括指令的地址、数据的地址。要执行一条指令，首先就需要回答“指令在哪？”的问题，在执行一条指令之后，需要回答下一条指令的地址在哪的问题？程序的三种结构：顺序、分支、循环，都涉及指令地址变迁的问题。无条件转移、简单条件转移、比较转移、循环、调用子程序、从子程序返回、中断和中断返回等等指令都涉及到改变待执行指令的地址。对于数据而言，可存放在各种各样的变量之中，可以通过多种寻址方式来得到操作数的地址。通过不同寻址方式的对比，可以更深刻的理解得到地址的方法，从而更灵活的编写程序。

（6）掌握语言的核心要素

对于语言而言，不论是自然语言还是计算机程序设计语言，涉及的核心内容就是：语法、语义、语用。要掌握语句的语法规则，并遵循语法规则，理解语句完成的功能，并在实际应用中，能够灵活的运用不同的语句来组装成程序。

在汇编语言中，重中之重是“地址”。从内存物理地址编址方法到逻辑地址与物理地址之间的映射，从源和目的操作数的寻址方式到语句中地址表达式的表述，各个数据之间的位置关系，乃至指令的地址，都是围绕地址这一核心展开的。掌握地址的多种表达方式，掌握不同变量之间或者某一结构变量中各成员之间的地址的相互关系，地址类型转换，就可以很灵巧的写出各种程序。

（7）算法细化，牢记“拆”字

对于一条 C 语句“z = 3 * x + 6 * y+ 4*8;”，站在 CPU 的角度来看，显示不能一步完成该语句的功能，它需要做多项工作。首先需要从内存中将变量 x 中的内容取到 CPU 中来，然后做一次乘法，并记住乘法的结果；之后取变量 y 中的内容，做第二次乘法，之后，还要做加法，最后将结果送到变量 z 中。汇编语言指令不能像高级程序设计语言那样一条语句完成较多的功能，而一次只能完成一个很简单的操作，由简单操作组合完成较复杂的功能。因此，在编写汇编语言源程序时，要学会“拆”，将要完成的功能细分成基本功能。不要受到高级语言程序的影响，企图在一条汇编语言中完成很复杂的功能。

（8）在实验中深化对所学知识的理解

在进行实验时，要仔细和小心，因为每条指令都只会引起小的变化，容易被忽略。在调试程序前，应预判指令和程序段的执行结果，即应用所学理论进行分析，通过实验进行验证。若分析结果和实验看到的结果不一致，则说明出现了一些问题。通过进一步的分析，查找原因，就可以促进对所学知识的理解和灵活应用。

1．4 汇编语言源程序举例

【例 1.1】 求 1+2+3+.....+100 的和，然后显示该和。

为了便于阅读和理解程序，在程序中写了一些注释。在 Intel x86 汇编语言源程序中，以西文的分号开头直到这一行结束都是注释。

```
.686P
.model flat, c
 ExitProcess proto stdcall :dword
 includelib  kernel32.lib
 printf      proto c :vararg
 includelib  libcmt.lib
 includelib  legacy_stdio_definitions.lib
.data
 lpFmt  db  "%d",0ah, 0dh, 0
.stack  200
.code
 main proc
      ; 下面7行语句实现的功能 eax=0; for (ebx=1;ebx<=100;ebx++) eax=eax+ebx;
    mov  eax, 0    ;  (eax)=0  eax 是 CPU 中的一个寄存器，用于存放累加和
    mov  ebx, 1    ;  (ebx)=1  ebx 是 CPU 中的一个寄存器，用于指示当前的加数
lp: cmp  ebx, 100  ; 连续两条指令，等同于 if (ebx>100) goto exit,循环结束
    jg   exit
    add  eax, ebx   ;  (eax) = (eax) + (ebx)
    inc  ebx        ;  (ebx) = (ebx) +1
    jmp  lp         ;  goto lp   无条件跳转到 lp 处执行
exit:
   invoke printf,offset lpFmt,eax  ; 调用 printf 函数，显示结果
   invoke ExitProcess, 0           ; 调用 ExitProcess，返回操作系统
main  endp
end
```

对于该程序，使用 19.1 节介绍的操作方法，可生成 exe 文件，运行该程序，将显示 5050。

下面首先介绍该算法的实现思想。若简单的将程序中出现的寄存器理解成 C 语言中变量（不需要编程者定义，在 CPU 中），用 C 语言语句描述出来核心程序段如下。

```
     eax= 0;      //  eax 用于存放累加和
     ebx= 1;      //  ebx 用于指示当前的加数
 lp:
     if  (ebx>100)  goto  exit;   // (ebx)>100 时循环结束
     eax = eax+ ebx;
     ebx = ebx +1;
```

```
    goto  lp    // 无条件跳转到 lp 处执行
exit:
```

该例是一个 Win32 控制台的程序，它给出了汇编语言源程序的基本结构。

首先使用处理器选择伪指令“.686P”，即在该程序中可以使用 Pentium Pro 微处理器所支持的指令。在 6.2 节给出了可以采用的处理器选择伪指令。

“.model”表示程序采用的存储模型说明伪指令，此处采用的存储模型（即内存管理模式）是 flat，代码和数据全部放在同一个 4G 空间内，段的大小是 32 位（4G 字节），这是一个 32 位段程序。flat 之后的 c 指明了所采用的“语言类型”，用于明确函数命名、调用和返回的方法。c 类型表示采用堆栈来传递参数，并且函数调用中最右边的参数最先入栈、最左边的最后入栈；在调用函数中清除参数所占的空间。在 6.2 节给出了 model 伪指令后可用的其他选项。

ExitProcess 是 Windows 操作系统提供的 API 函数，其功能是终止一个进程的运行，返回操作系统。实现 ExitProcss 时用语言类型是 stdcall，在函数说明时必须与其实现保持一致。该函数的实现是在库 kernel32.lib 中，使用 includelib 包含该库。

printf 是 C 语言程序中最常见的一个函数，用于格式化输出一串信息，其语言类型为 c。“includelib libcmt.lib”和“includelib legacy_stdio_definitions.lib”指明了 printf 函数实现所用到的库。

“.data”是数据段定义伪指令，在该段中定义了一个全局变量 lpFmt，该定量中赋的初值是“″%d″,0ah, 0dh, 0”，等同 C 语言中的串“%d\n”。注意，在 data 段中定义的所有变量都是全局变量，而在函数（子程序）中定义的变量是局部变量。

“.stack”是堆栈段定义伪指令，其后的 200，表示堆栈段的大小为 200 个字节。

“.code”是代码段定义伪指令。

在汇编语句源程序中“data”、“stack”、“code”都是关键字，分别对应数据段、堆栈段、代码段，它们对应一个段的开始，同时表明前一个段的结束。

“main proc … main endp”是子程序。在程序的最后有一条伪指令“end”表示程序结束了，编译器看到该伪指令后，就不会再往下编译程序。由于使用的是 main，自动将 main 作为程序的入口点，这与 C 语言程序中缺省的是从 main 处开始执行一致。

从上面的汇编源程序可以看出，一个源程序由一系列语句组成，每个语句占一行，最后均以回车作结束。每一个语句一般由 4 个部分组成，这 4 个部分按照一定的规则，分别写在一个语句的 4 个区域内，各区域之间用空格或制表符（TAB）隔开。语句的一般格式为：

［名字］ 操作符 ［操作数或地址］ ;［注释］

实际上，并不是每个语句都需要这 4 个部分，但操作符是必不可少的，其它部分为可选项，因而用方括号括起来。名字由字母 A～Z、a～z、数字 0～9、或者一些特殊字符如 _（下划线）、?（问号）、@、$等组成的字符串，但该字符串不能以数字作开始字符。用户所定义的名字一定不能与系统中的保留关键字如：指令操作码、寄存器名、汇编程序规定的运算符号等同名。在一个程序中，名字不可以重复定义。标号也是一个名字，但在标号后有西文的冒号“:”。

注释均以西文的分号开始，它可占一行或多行，也可放在一条语句的后面。从分号开始到一行的结束都是注释。注释如果放在一条语句的后面往往是用来说明该语句使用的目的、功能或在逻辑流程中的作用。如果放在程序的开始或中间一般是用来说明该程序的功能、基本设计思想和用法。使用注释可使程序便于阅读、修改和交流推广。汇编程序对注释部分不作处理，也不产生目标代码，只作为文本随源程序一起输入、存储或输出，对程序的执行也无任何影响。在一般情况下，注释可用英文或拼音输入，如果系统是使用的中文操作系统，注释也可用中文输入。但要注意，除注释内容和字符串定义外，其它内容（包括分号在内的

标点符号）应该是标准的 ASCII 字符。

操作符为语句的核心成分，它表示了该语句的操作类型。当操作符是机器指令的助记符时，该语句是机器指令语句，由于机器指令语句基本上是与机器的指令一一对应，在本书的后面，我们也将它简称为机器指令或指令；如果是伪指令的助记符，则为伪指令语句；如果是宏定义名，则该语句是宏指令语句。

有些机器指令语句后面是没有操作数的，它的操作对象是固定的。有些是带一个操作数或操作数地址，例如，inc ebx。有些指令可带两个操作数地址，它们可以是寄存器名、存储单元地址等，之间用西文的逗号隔开，这是双操作数指令。例如：

```
            add   eax, ebx
目的操作数地址 ←┘   └→ 源操作数地址
```

一般将目的操作数地址简称为目的地址，用抽象的符号 opd 表示；将源操作数地址简称源地址，用符号 ops 表示。目的操作数和源操作数由 opd 和 ops 所示的寻址方式获取。“add opd, ops”的功能为：

(opd)＋(ops)→opd

对应具体的指令“add eax, ebx”，实现的功能为 (eax)＋(ebx)→eax。

操作结束后运算结果保存在目的地址中，源地址的内容并不改变。

在编写 C 语言程序时，符号是一般是区分大小的。在汇编语言源程序中，在缺省状态下，对于程序内自己定义的变量名、标号名、子程序名在缺省状态下是不区分大小写的。例如，数据段中定义了变量 lpFmt，在代码段中，可以写成 LPFMT，也可以写成 lpfmt，它们是等同的。如果要区分大小写，可以在程序开头增加一行语句“option casemap : none”。当然，对于系统所使用的关键字，例如指令助记符、寄存器名、“data”、“code”、“end”等等，不论有无“option casemap : none”，都是不区分大小写的。因此，程序中 EAX 与 eax 是等同的、mov 与 MOV 等同。在程序中，调用的外部函数，如 printf、ExitProcess，应与外部函数的定义保持相同，不能随意使用大小写。

【例 1.2】输入 5 个有符号数到一个数组缓冲区中，然后求它们的最大值并显示该最大值。

实现该功能的算法思想很简单，先用循环的方法输入 5 个数，然后将第 1 个数放在 eax 中，再次采用循环的方法，后面的数依次和 eax 比较大小，若后面的数大，则将其放到 eax 中。下面直接给出了程序。

```
.686P
.model flat, c
  ExitProcess proto stdcall :dword
  includelib  kernel32.lib
  printf      proto c :ptr sbyte, :vararg
  scanf       proto c :ptr sbyte, :vararg
 includelib   libcmt.lib
 includelib   legacy_stdio_definitions.lib
.data
  lpFmt   db  "%d",0ah, 0dh, 0
  buf     db  "%d", 0
  x       dd  5 dup(0)     ;  int x[5];
.stack 200
.code
main  proc
```

```
    ; 输入 5 个数   ebx=0;
    ;               do { scanf("%d",&x[ebx*4]); ebx++; }while (ebx!=5)
    mov   ebx, 0
input_5num:
    invoke scanf, offset buf, addr [x+ebx*4]
    inc    ebx
    cmp    ebx, 5
    jne    input_5num
    ; 求 5 个数中的最大数。先将第一个数作为最大数，放在 eax 中。
    ;    eax=x[0];
    ;    for (ebx=1;ebx<=4;ebx++)
    ;       if (eax<x[ebx*4])  eax = x[ebx*4];
    mov  eax, x[0]       ; eax = x[0]
    mov  ebx, 1          ; ebx = 1
lp: cmp  ebx, 4          ; if (ebx>4) goto exit
    jg   exit            ; 结束找整个最大数的循环
    cmp  eax, x[ebx*4]   ; (eax)比当前数大或者相等，则不要做任何处理
    jge  next
    mov  eax, x[ebx*4]   ; eax = x[ebx*4]
next:
    add  ebx, 1          ; ebx = ebx+1
    jmp  lp              ; goto lp
exit:
    invoke printf, offset lpFmt,eax
    invoke ExitProcess, 0
main  endp
end
```

在例 1.2 的程序中，对 printf 的定义稍有变化，这里给的原型说明更严谨一些。C 语言中，有 int printf(const char *format， ...)；即 printf 第一个参数是一个串指针，后面才是可变个数的参数。:ptr sbyte 即表示串指针，:vararg 表示可变参数。

从上例可以很清楚地看到两种循环语句的执行过程，也看到数组 x 的定义和访问方式。与 C 语言中 x[ebx]的访问不同，在机器语言中是按字节对地址编码的，由于定义的是一个双字操作数组，每个元素占 4 个字节，故 ebx 要乘 4，后面的章节中还会详细的讲解。有兴趣的读者可以用 C 语言写一个程序完成相同的功能，然后调试时显示反汇编语句，就可以看到对于 x[ebx]，会自动的将 ebx 乘以 4。也就是说，编译器看到定义的 int 类型数组后，知道一个元素与下一个元素之间的距离是 4 个字节。数组的第 ebx 个元素，与数组起始地址相距 ebx*4 个字节，这一转换工作就由编译器来完成了，而写汇编语言程序时，就要准确的给出访问单元的地址。

1．5 计算机中信息编码的奥秘

在学习汇编语言的时候，需要深刻理解计算机工作的本质。本节探索计算机世界的信息编码以及计算机世界中的信息与外部世界信息的一些对应关系。

计算机世界是 0 和 1 的世界，计算机之外的丰富多彩的现实世界，都要编码成 0-1 串存

储到计算机中。而在解析 0-1 串的含义时候，要依赖于解析时的场景动态变化。通过解码，由计算机世界回到现实世界。0 和 1 组成的串是计算机内部存储和加工的对象，其外部表象是花花绿绿的世界。

围绕计算机世界是 0-1 世界这一论断，读者可以探索以下问题。为什么信息表达用 0 和 1 来编码？现实世界中的信息又如何编码转换成 0-1 串？计算机中所有的信息都是 0-1 组成的串，计算机又是如何解读这些串的呢？计算机内部单调平凡的 0-1 串又是如何在我们面前展现丰富多彩的画面？

1、计算机中所有信息都是以 0-1 串的形式存储的

在阅读了 1.1 节后，是否会产生一个误会，以为只有执行程序是二进制编码。实际上，源程序也是二进制编码。计算机中存储的对象，不论是图像文件、视频文件、声音文件，还是 WORD 文件、PPT 文件、程序文件（源程序文件、执行程序文件）、动态库等等，都是由 0 和 1 编码组成的文件。换句话说，在计算机上存储的文件、在网络上传输的信息，都是由 0 和 1 编码而成。

有兴趣的读者，可以用文本编辑器 UltraEdit、软件开发集成环境 Microsoft Visual Studio 等软件打开一个文件来体验一下。只是在打开文件时，注意选择文件的打开方式为“二进制编辑器”或者“以二进制的形式打开”。当然，展现在我们面前的是十六进制文件，这只是让人更容易读一些，其本质是二进制文件。

2、为何使用 0-1 编码

计算机内部硬件能够识别的只有 0 和 1，也就是二进制。因为二进制只有两种状态。计算最底层的硬件通过多种方式来支持两种状态的识别和处理，就像灯泡的“亮”和“灭”、电压的“有”和“无”、磁性材料的“N”和“S”、电极的“正电”和“负电”、电平的高和低等。

3、编码是有一定规范的

任何文件中存储的内容都是有一定编码规则的，如果不知道其编码规则，就无法解析 0 和 1 组成的串所表达的含义。

以二进制形式打开 1.1 节中的 C 语言源程序 c_example.c，可以看到如图 1.6 所示的信息。

c_example.c

```
00000000  23 69 6E 63 6C 75 64 65  20 3C 73 74 64 69 6F 2E  #include <stdio.
00000010  68 3E 0D 0A 0D 0A 69 6E  74 20 6D 61 69 6E 28 69  h>....int main(i
00000020  6E 74 20 61 72 67 63 2C  20 63 68 61 72 2A 20 61  nt argc, char* a
00000030  72 67 76 5B 5D 29 0D 0A  7B 0D 0A 09 69 6E 74 20  rgv[])..{...int
00000040  20 78 2C 20 79 2C 20 7A  3B 0D 0A 09 78 20 3D 20   x, y, z;...x =
00000050  31 30 3B 0D 0A 09 79 20  3D 20 32 30 3B 0D 0A 09  10;...y = 20;...
00000060  7A 20 3D 20 33 20 2A 20  78 20 2B 20 36 20 2A 20  z = 3 * x + 6 *
00000070  79 2B 20 34 2A 38 3B 0D  0A 09 70 72 69 6E 74 66  y+ 4*8;...printf
00000080  28 22 33 2A 25 64 2B 36  2A 25 64 2B 34 2A 38 3D  ("3*%d+6*%d+4*8=
00000090  25 64 5C 6E 22 2C 78 2C  79 2C 7A 29 3B 0D 0A 09  %d\n",x,y,z);...
000000a0  72 65 74 75 72 6E 20 30  3B 0D 0A 7D 0D 0A        return 0;..}..
```

图 1.6 以二进制形式显示文件 c_example.c

在窗口的右半部分，是以字符形式显示了 c_example.c；在中间就是以二进制形式显示了该文件的内容。这些信息是按照 ASCII 码（American Standard Code for Information Interchange，美国信息互换标准代码）对各个字符顺序编码得到的。在书的附录中给出了字符的 ASCII 表。

在 ASCII 表中，可以看到英文小写字母 a 到 z 的 ASCII 码是顺序的编排在一起，从 97（61H）到 122（7AH）。如果给定一个字符是小写字母，只要用该字符的 ASCII 码减去字母 a 的 ASCII 码，也就可以知道它是小写字母表中的第几个字母。为了减少记忆和避免出错，在 C 语言程序和汇编语言源程序中，可以直接使用'a'的表达形式来代替 97。由编译器完成

将'a'翻译成 97 的工作，例如 x-'a'等价于 x-97。在 ASCII 码表中也可以看到，大写字母 A 到 Z 的 ASCII 码同样是顺序的编排在一起，从 65（41H）到 90（5AH）。大小写字母之间的转换可以利用该规律完成。例如小写字母 x 转换成大写字母，可以写成 x-20H。当然，也可以表达为 x-'a'+'A'。数字 0~9 的 ASCII 码为 30H~39H。大小写字母以及数字字符的编码在汇编语言程序设计中是常用的。

除了对字符信息的 ASCII 编码外，常见的字符编码规范还有 Unicode、UTF-8 等等。Unicode 是国际组织制定的可以容纳世界上所有文字和符号的字符编码方案。Unicode 字符集可以简写为 UCS（Unicode Character Set）。早期的 Unicode 标准有 UCS-2、UCS-4。UCS-2 用两个字节编码，UCS-4 用 4 个字节编码。有兴趣的读者可以通过互联网或者其他方式获得 Unicode、UTF-8 等更详细的知识。通过了解编码规范，也可以了解为何出现“中文乱码”以及如何解决这一问题等知识。

除了文本文件外，还有多种类型的文件，如 WORD 文件、PDF 文件、图像文件、视频文件、可执行文件等等，各自都有自己的编码规范。当然，在我们的日常生活中，遵循某种编码规范的信息随处可见，例如，身份证号就有严格的编码规则，从身份证号上不但可以看到出生的年月日，也可以看到性别，所属的省、市、区县等信息。正是有了各种规范，信息才能被正确的理解。

4、如何知道一个文件采用的编码规范

文件的内容都是二进制文件，是由 0-1 组成的串，计算机如何知道一个文件采用的编码规范，从而正确的对文件进行解读呢？

计算机中采用的方法是，在文件的一个特殊地方，通常在文件开头存放文件所采用的编码规范。例如：Unicode 的文件头是 FF FE；UTF-8 的文件头是 EF BB；JPEG 的文件头是 FF D8 FF；BMP 的文件头是 42 4D（即“BM”的 ASCII 编码串）；Word 文件头（docx）：50 4B 03 04；PDF 文件头：25 50 44 46 （即“%PDF”的 ASCII 编码串）等等。

有兴趣的读者可以在互联网上搜索，“如何获知一个文件的编码方式”、“如何将一个文本文件保存为 Unicode 编码”，了解更多的有关编码的知识。

5、指令是如何进行译码的？

在一个程序中，所有的数据是 0-1 串，所有的指令也是 0-1 串，如何区分是数据还是指令呢？在一条指令中，所要进行的操作运算符以及操作数或者操作数的地址，都是用 0 和 1 编码的。如何区分一个 0-1 串是表示操作，还是操作数的地址呢？或者是一个操作数地址表达式中的一部分呢？要解答这些问题就需要知道 X86 指令格式及编码规则。本书在第 4 章中将介绍 X86 的寻址方式，之后的第 9 节将详细介绍指令的编码规则。

总之，计算机世界是 0 和 1 世界，采用一定的编码规则，将现实世界的信息编码成 0-1 串存放到计算机中，并进行加工和处理。根据文件中给定的或者默认的编码规范，通过解码又可以回到现实世界。

1．6 书中使用符号的说明

在后面的学习中，需要引用以下一些符号，在整本书中都将遵循这些写法。

（…）表示地址“…”中的内容。

例如，假设变量 x 的偏移地址为 0x0040ff08，其对应的存储单元中的内容为 10，则表示为(0x0040ff08)＝10，或者(x)=10；寄存器 ebp 的内容为 0x0040ff10，则表示为(ebp)＝0x0040ff10。

[…] 表示以地址“…”中的内容为偏移地址。

例如，(ebp)＝0x0040ff10，而(0x0040ff08)＝10，则([ebp-8])表示以 ebp 的内容

减去 8 位为偏移地址，在该偏移地址中存放的数据。此处的([ebp-8])=10。

EA 某一存储单元的偏移地址（Effective Address），即指该存储单元到它所在段段首址的字节距离。

PA 某一存储单元的物理地址(Physical Address)。

R 指某寄存器(Register)的名字。

SR:[R] 指二维的逻辑地址（指针）。其中，SR 为段寄存器名，[R]表示某寄存器 R 为指示器，存放着该段内某一存储单元的偏移地址。

OPD 目的地址，即目的操作数存放的偏移地址。

OPS 源地址，即源操作数存放的偏移地址。

→ 表示传送。

例如，1234H→bx 表示将操作数 1234H 传送到寄存器 bx 中,即 mov bx,1234H;

指令 mov dword ptr x,0AH, 表示为 0AH →x

若 x 的地址是 0x0040ff08，该指令的功能也可以表示为 ：0AH →0x0040ff08。

↑(sp)/(esp) 其中：↑(esp)表示从 32 位堆栈段出栈(弹出)；

↑(sp) 表示从 16 位堆栈段弹出。

↓(sp)/(esp) 其中：↓(esp) 表示向 32 位堆栈段进栈(压入)。

↓(sp) 表示向 16 位堆栈段压入。

∧ 表示逻辑乘。

∨ 表示逻辑加。

⊕ 表示按位加。

$\overset{\frown}{\times\times\times}$ 表示需要对数“×××”作求补运算。

$\overline{\times\times\times}$ 表示需要对数“×××”作求反运算。

××H 表示数“××”为十六进制数。在 C 程序中，十六进制数以 0x 开头。

×× 表示数“××”为十进制数。

××B 表示数“××”为二进制数。

××BCD 表示数“××”为 BCD 码。

EFLAGS 表示标志寄存器。

/ 表示或者。

$\underline{\times\times\times}$ 表示横线上面的内容“×××”是通过键盘输入的。

例如，有以下书写形式：

* $\underline{\text{ABC}}$ ↙

其中，“* ”下面未加横线，表示是由显示器显示的内容；而“$\underline{\text{ABC}}$”下面有横线，说明是由键盘输入的内容；“↙”为行终止符，用以表示一行内容的结束，在输入时，是通过敲键盘上的回车键实现的。

习题 1

1.1 什么是机器语言？什么是机器语言程序？

1.2 不同 CPU 的机器语言是否相同？为什么本书叫做 X86 汇编语言程序设计？

1.3 为什么在指令编码中，所含的成份是变量的地址而不是变量中的值？

1.4 为什么设计机器指令时，一条指令只能完成一点简单的功能？这样设计的优点是什么？

1.5 什么是汇编语言？什么是汇编语言源程序？

1.6 什么是汇编？什么是反汇编？什么是伪指令？
1.7 机器语言、汇编语言、高级程序设计语言的关系是什么？
1.8 在什么情况下，需要使用汇编语言编写程序？
1.9 在什么情况下，别无选择只有阅读汇编语言程序？
1.10 什么是符号地址？

上机实践 1

1.1 自学第十九章上机操作，熟悉在 Visual Studio 2019 平台开发一个汇编语言源程序的方法和操作步骤，能够对例 1.1 和例 1.2 生成可执行程序并运行。
1.2 自学第十九章上机操作，熟悉在 Visual Studio 2019 平台开发一个 C 语言程序，包括程序调试，学习观察内存、寄存器、变量的值及变量的地址等，学习调试时用反汇编观察生成的指令。
1.3 调试 1.1 节中的程序 c_example.c，观察变量 x, y, z 的地址各是多少？在反汇编指令中，是如何表示这三个变量的地址的？

思考题

1.1 在 C 语言程序中，变量的定义语句和对变量处理的语句可以写在一起，但是调试时发现变量的地址与之后的处理指令的地址相距很远。为什么要将变量的空间分配与指令分隔开呢？
1.2 在 C 语言程序中，看不到数据段、堆栈段、代码段等分段的概念。为什么在汇编语言程序（或者机器语言程序）中，需要将数据和代码分开放在不同的段中？