第一章 绪 论

§1.1 汇编语言程序设计的一般概念

世界上第一部电子计算机诞生于本世纪四十年代中期，至今已 40 多年，在这期间它以惊人的速度得到发展，现在我们社会中无论哪个行业、部门、地区到处可见计算机的应用，直接使用或操作计算机的人已日益增多。在应用中，有的是按照应用程序规定的操作步骤使用计算机；有的学会某种程序设计高级语言（如 BASIC、FORTRAN、PASCAL、COBOL、C 等），使他们能编制程序，与计算机通信。但是，这其中许多人对计算机是如何工作的却了解不深。

从高级语言程序设计的观点来观察一台计算机，可能认为这台计算机就是执行高级语言各语句及其功能的机器。但是一台计算机实际上是执行由中央处理器（CPU）所提供的最基本功能——机器指令。机器指令是以代码的形式表示的，这对编制程序和阅读程序是相当困难的。而汇编语言程序是把由机器指令组成的机器语言程序"符号化"，并与机器语言程序一一对应。例如要完成二个数相加后再减去第三个数的运算，用 BASIC 语言可编制如下程序：

```
10      READ    A,B,C
20      LET     S=A+B-C
30      DATA    14,26,13
40      END
```

同样的运算，用 8086/8088 MASM 汇编语言可编制程序：

```
DATA    SEGMENT
A       DB      14              ;  数据  A
B       DB      26              ;  数据  B
C       DB      13              ;  数据  C
S       DB      0               ;  存放结果单元
DATA    ENDS
STACK1  SEGMENT  PARA  STACK
        DW      20H  DUP(0)
STACK1  ENDS
COSEG   SEGMENT
        ASSUME  CS:COSEG,DS:DATA,SS:STACK1
START:  MOV     AX,DATA
        MOV     DS,AX
        MOV     AL,A            ;取第一个数
        ADD     AL,B            ;计算二数之和
        SUB     AL,C            ;减去第三个数
        MOV     S,AL            ;存放运算结果
        HLT
COSEG   ENDS
        END     START
```

上述汇编语言程序对应的机器代码（即机器语言程序）如图 1.1 所示。

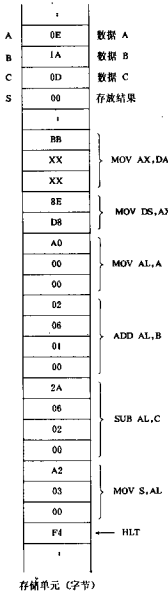


图1.1 机器语言程序

从上述例子可以看出：用汇编语言编制程序虽然比用高级语言编制程序"麻烦"些，"琐碎"些，但是比用机器语言编制程序要方便得多，易于学习、记忆、修改。用汇编语言编制的程序（这样的程序叫汇编语言源程序）仍然不能直接由计算机执行，亦必须经过"汇编"（即翻译），转换成机器语言程序，再由计算机执行。

由于每种计算机的设计者有不同的设计思想，不同的应用目的。因此，每种计算机有它自己机器的指令系统，相应的有自己的机器语言和汇编语言。为了学习、使用某种计算机的汇编语言就必须首先熟悉那种计算机。这里所说的"熟悉那种计算机"并非指组成计算机系统的全部硬件，而是我们用汇编语言编制程序时要涉及到的那些硬件内部结构和功能。一台计算机能执行的机器语言程序，主要决定于组成这台计算机的中央处理器CPU。因此，我们需要了解并熟悉计算机的内部结构主要是指CPU的功能结构：有多少个寄存器，各有什么用途；CPU是怎样访问存储器的（例如是怎样形成地址的）；在进行输入/输出操作时有些什么工作方式等等。本书是以IBM PC微型计算机及其汇编语言为例，它的CPU可以用Intel 8086微处理器，也可用Intel 8088微处理器。因为8086和8088有完全相同的指令系统，在软件上完全兼容。对于初学者应该明了，通过一个具体的计算机学习并掌握它的汇编语言程序设计的基本原理、方法和技巧，这对于今后在其他计算机上进行汇编语言程序设计是非常有益的。

§1.2 为什么要学习和使用汇编语言

凡是学过一种程序设计的高级语言，都会有高级语言"易学好用"的感觉，这是因为这些语言的语句是面向数学语言或自然语言的，因此容易接受、掌握。相对来说用汇编语言编制程序比用高级语言要困难些。既然如此，为什么至今还要学习和使用汇编语言呢？

1. 学习和使用汇编语言可以从根本上认识、理解计算机的工作过程，因为一台计算机执行一个任务，归根到底就是执行一个计算机机器语言程序。通过用汇编语言编制程序，可以更清楚地了解计算机是怎样完成各种复杂的工作。在此基础上，程序设计人员更能充分地利用机器硬件的全部功能，发挥机器长处。这就如同你编制的程序在计算机上运行时，它调动并控制着机器中每个部件和电路。

2. 现在计算机系统中，某些功能仍然靠汇编语言程序来实现的。例如机器的自检，系统的初始化，实际的输入输出设备的操作，至今仍然是用汇编语言编制的程序来完成。

3. 汇编语言程序的效率通常高于高级语言程序。这里的"效率"是指程序的目标代码的长短和程序运行的速度。所以在节省内存空间和提高程序运行速度是重要指标场合，如实时过程控制，常常是用汇编语言来编制程序。

鉴于以上理由，现在许多高级语言都设置有与汇编语言程序接口的功能，以便于用户用汇编语言编制某些子程序，完成与机器联系紧密的特定功能，提高高级语言程序的效率。

§1.3 计算机中数和字符的表示

这里对计算机中信息的表达不作详细讲解，但是鉴于用汇编语言编程的要求，在此扼要介绍，学习汇编语言程序设计时所必要的有关基础知识。

1. 数制及其数制间的转换

目前计算机内部总是采用二进制数(Binary)进行操作和运算的。但是，在用汇编语言编制源程序时，为了书写方便，易于检查，常常是不采用二进制数。如在 IBM PC 微机的汇编语言程序中，主要是使用十进制数(Decimal)，十六进制数(Hexadecimal)，当然也可以使用二进制数。从汇编语言源程序转换为目标程序时，程序中的十进制数或十六进制数均由汇编程序(Assembler)自动转换成二进制数。但是，程序设计人员必需熟练地掌握这些数制间的转换。

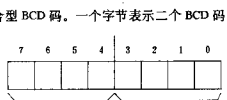
2. 码制

为了在计算机中能表示出正数和负数，每个字节或字的最高位设置为符号位。当符号位＝0时，该数为正数。当符号位＝1时，该数为负数。为了便于计算机运算，通常采用原码，反码，补码，移码来表示带符号数。IBM PC 微机汇编语言程序中是采用补码(2's Complement)表示的。因此，对于一个字节(八位二进制数)的带符号数，可表示的最大正数为 01111111B 或 7FH，即十进制数＋127，可表示的最大负数为 10000000B 或 80H，即十进制数－128。对于一个字(十六位二进制数)的带符号数，可表示的最大正数为十进制数＋32767，最大的负数为十进制数－32768。为了表示一个数据是什么进制数，以后我们在数字后加字母 'B' 表示二进制数；加字母 'H' 表示十六进制数；加字母 'D' 或不加字母表示十进制数。

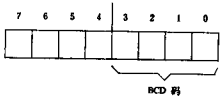
3. BCD 码

二进制数对计算机的操作和运算是十分方便容易的。但是人们在日常生活中较习惯于使用十进制数。为了便于计算机直接使用十进制数，产生了用二进制编码的十进制数，即 BCD 码(Binary Coded Decimal)。一位十进制数用四位二进制数表示。这四位二进制数通常用 8421 编码方式，其中四位二进制数 0000～1001 表示十进制数 0～9，其余的四位二进制数 1010～1111(即十六进制数的 A～F)不用。在 IBM PC 微机中，当使用 BCD 码时，可以有二种 BCD 码的存储方式：

第一种方式——组合型 BCD 码。一个字节表示二个 BCD 码：

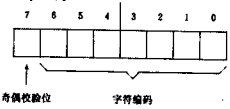


第二种方式——非组合型 BCD 码。一个字节表示一个 BCD 码：



4. 字符编码

现在计算机中通常采用的字符编码是 ASCII 码(American Standard Code for Information Interchange)。标准的 ASCII 码(见附录 A)是一个字节中用七位二进制表示字符编码，其中一位(最高位)是奇偶校验位(Parity bit)。



标准的 ASCII 码共有 128 个字符，可分为二类：非打印的 ASCII 码和可打印的 ASCII 码。

. 非打印 ASCII 码：这类编码是用于控制性代码，共 33 个。如 BEL(响铃，07H)，DEL(删除，7FH)，CR(回车，0DH)，LF(换行，0AH)等。

. 可打印 ASCII 码：共有 95 个。例如：

数字 0～9 的编码	30H～39H
大写字母 A～Z 的编码	41H～5AH
小写字母 a～z 的编码	61H～7AH
空格(space)的编码	20H

习　题

1. 试根据自己使用计算机的经历，例举几个必须使用或最好是使用汇编语言编制程序的事例。

2. 试完成下列数制间的转换：

(1)十进制数转换为八位二进制数：

100 ， 56 ， 111 ， 120 ， 70

(2)八位二进制数(无符号数)转换为十进制数：

01010101 ， 10101010 ， 11110000 ， 00001111

(3)十进制数转换为十六进制数：

40 ， 80 ， 105 ， 114 ， 207

3. 试把下面用补码表示的二进制数转换为对应的十进制数真值：

01111000 ， 11011001 ， 10000001

10001000 ， 00100111 ， 11110000

4. 由键盘输入字符通常都是以该字符 ASCII 码形式表示的。若现在从键盘上输入十六进制数 0～F，那么应如何处理才能把十六进制数字符转换为四位二进制数 0000～1111。

第二章　IBM PC 微型计算机

§2.1　IBM PC 微型计算机基本结构

IBM PC 微型计算机是美国国际商业机器公司(International Business Machine Corp)于1981年8月正式推出的个人计算机(Personal Computer)。当今 IBM PC 系列微机及其兼容机已成为国际上最流行的十六位微型计算机。它不仅有较先进的系统，而且有相当丰富的软件支持，可以广泛地应用于科学计算，事务管理，过程控制等诸方面。

一台电于计算机通常由五大部分组成：运算器、存储器、输入设备、输出设备和控制器，现在的微型计算机也基本上是由这五大部分构成。其中把运算器和控制器两部分集成在一个芯片上，称为微处理器(Microprocessor)，即中央处理器 CPU(Central Processing unit)。微机的一般结构如图 2.1 所示。

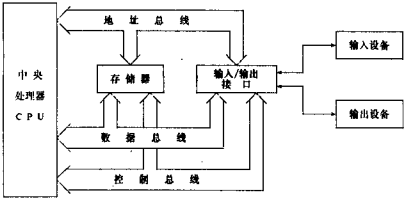


图2.1　微型计算机一般结构

IBM PC 微机的系统配置如下：

1. CPU 采用 Intel 8086 或 Intel 8088。8088 是一个准十六位微处理器，即内部结构是十六位的，而外部的数据总线是八位，在一个总线周期内只能吞吐一个字节，而 8086 的内部结构与外部总线均是十六位，在一个总线周期内可吞吐一字，但是它们两个的体系结构是类似的。指令系统，指令编码格式，寻址方式都完全相同。在软件上是完全兼容的。

Intel 8086/8088 CPU 共有 14 个十六位寄存器(如图 2.2 所示)。分别为：通用寄存器(8个)，控制寄存器(2个)，段寄存器(4个)。

Intel 8086/8088CPU 共有 20 条地址线，可直接寻址 2^{20} = 1M 字节的内存空间(即1048576个字节)。I/O 端口可扩展到 64K(即 0~65535)个端口编号进行输入输出。

2. 通常 IBM PC 系列微机的大板上装有 40K～48K 字节的 ROM 或 EPROM，64KB 或 128KB 的 RAM，用户可扩充到 640KB。

图2.2 Intel 8086/8088 内部寄存器

3. 系统时钟频率为 4.77MHz。

4. 外存：通常配置有 1～2 个 5.25 英寸软盘驱动器和一个温氏(Wenchester)硬盘(10MB 或 20MB)。

5. 系统大板上配置有 5～8 个扩展插座。系统接有键盘一个，单色或彩色显示器一个，打印机一台。

§2.2 Intel 8086/8088 微处理器的功能结构

计算机执行一个程序就是由 CPU 逐条执行组成该程序的指令序列，CPU 基本上是重复完成：

- 从存储器中取指令；
- 执行指令。

上述重复步骤，在以前的微机(如八位微机)或小型计算机(如 PDP－11 系列机)上是以串行方式进行的。如图 2.3(a)所示，其中取指令必须访问存储器，执行指令有时访问存储器，有时可以不访问存储器。凡是访问存储器就必须占用 CPU 的外部地址总线和数据总线，因此外部总线的"忙""闲"不均，而且每当总线"忙"时，CPU 内部又"闲"。显然，这种串行工作方式影响了程序的执行速度。

Intel 8086/8088CPU 采用全新的体系结构形式——指令流水线结构(Instruction Pipeline)。它把访问存储器(含取指令，存取操作数等)与执行指令分成两个独立部件：总线接口单元 BIU(Bus Inteface Unit)和执行单元 EU(Execute Unit)，如图 2.4 所示。

执行单元 EU 的功能是从 BIU 的指令队列中取出指令代码，然后执行指令所规定的全部功能。如果在执行指令的过程中，需要向存储器或 I/O 传送数据时，EU 向 BIU 发出访问存储器或 I/O 的命令，并提供访问的数据和地址。

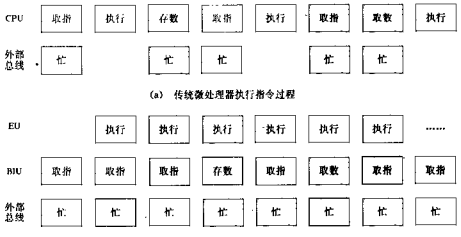


图2.3 微处理器执行指令过程

总线接口单元 BIU 负责 CPU 与存储器、I/O 的信息传送。具体功能是根据段寄存器 CS 和指令指针 IP 形成 20 位的物理地址，从存储器中取出指令，并暂存在指令队列中，等待 EU 取走并执行。如 EU 发出访问存储器或 I/O 端口的命令时，BIU 根据 EU 提供的数据和地址，进入外部总线周期，存取数据。

由此可知，EU 和 BIU 是既分工又合作的两个独立部件。它们的操作是并行的，分别完成不同的任务，因而大大加快了指令执行速度。从宏观来看，Intel 8086/8088 运行程序时，CPU 的执行过程大致如图 2.3(b)所示。

§2.3 Intel 8086/8088 CPU 寄存器结构及其用途

对于汇编语言程序设计者，CPU 中各寄存器、存储器和 I/O 端口是它们进行编程的基本活动"舞台"。而且大部分指令都是在寄存器中实现对操作数的预定功能。

一、通用寄存器(General Register)

这类寄存器共有 8 个，根据使用情况可分为三种：

1. **数据寄存器**(Data Register)

它包含 AX,BX,CX,DX 四个寄存器。这四个寄存器的每一个，既可作为十六位寄存器来使用，也可作为两个八位寄存器来使用。因此，这四个十六位寄存器也可以看成是八个独立的八位寄存器 AH,AL,BH,BL,CH,CL,DH,DL(见图 2.2)，它们分别由十六位寄存器的高八位和低八位构成。在程序中，这些寄存器(八位或十六位的)既可存放操作数，也可存放操作结果。在许多指令中，它们没有固定的应用，具有良好的通用特性，由编程人员任意选用。在多数情况下，选用这些寄存器必须在指令中指明。例如：

```
MOV   AX,BX      ;  把 BX 内容传送给 AX
ADD   CH,DH      ;  DH 和 CH 内容相加，结果送 CH
SAL   DL,1       ;  DL 内容算术左移一位
```

但有少数指令，虽然在指令中未指明，但隐含使用了某寄存器。例如，在循环指令(LOOP)中，

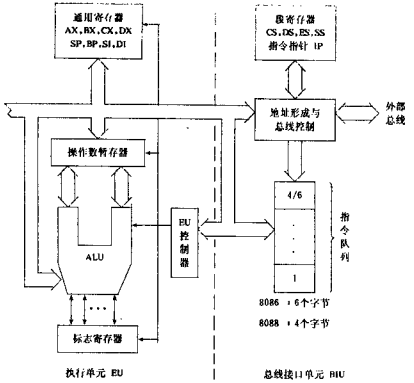


图2.4 Inter 8086/8088 CPU 功能结构框图

CX 作为循环次数计数器用。每执行一次 LOOP 指令，CX 内容减"1"，如(CX)≠0，则循环至 LOOP 指令目标地址处，否则循环结束，顺序执行下一条指令。在两个字节数相乘指令中，隐含使用了 AX(AL 存放一个操作数，AX 存放乘积)。在两个字相乘指令中，隐含使了 DX，AX 寄存器(AX 存放一个操作数，乘积存放在 DX：AX 中)。隐含使用寄存器的情况见表 2.1。有个别指令对寄存器有特定的应用，且又必须在指令中指明(非隐含性)。例如，移位类指令，当移位次数大于"1"时，应把移位次数事先送入 CL 寄存器，然后在移位指令中指定 CL 作为移位次数寄存器。如"SAR AX,CL","ROL DATA_BYTE,CL"等。

2. 指针寄存器(Pointer Register)

堆栈指针(SP)和基址指针(BP)通常用来作为十六位地址指针。由于 Intel 8086/8088 形成访问存储器 20 位的物理地址是由段的起始地址和偏移量组成(物理地址的形成将在§2.4 中详细讨论)，SP 和 BP 作地址指针时，就是存放这个偏移量。其中 SP 是特定指向堆栈段内的某一存储单元(字)的偏移量。当进行堆栈操作(压入或弹出)时，隐含使用的就是堆栈指针 SP。如果在指令中不另加说明，用 BP 作地址指针时，它亦是指向堆栈段内某一存储单元(字或字节)。

3. 变址寄存器(Index Register)

两个变址寄存器 SI、DI 均是十六位的，通常也被用来作地址指针。在变址寻址中，SI、DI 的内容作为段内偏移量的组成部分。在多数情况下，源和目的变址寄存器，可由用户随意选用。但是在串操作指令中，SI 和 DI 就有区别。源串操作数必须用 SI 提供偏移量，目的串操作数必须用 DI 提供偏移量。对于串操作指令，SI、DI 的作用不能互换。而且它们在指令中隐含

规定使用哪一个或两个。

表 2.1 通用寄存器的特定使用和隐含使用

寄存器	隐含/特定使用	隐含使用/特定使用
AL AX	① 在乘法指令中，存放乘数和乘积. ② 在除法指令中，存放被除数和商. ③ 在未组合 BCD 码运算的校正指令中. ④ 在某些串操作指令中(LODS,STOS,SCAS)中. ⑤ 在输入/输出指令中作数据寄存器.	隐 含 隐 含 隐 含 隐 含 特定使用
AH	在 LAHF 指令中作目的寄存器.	隐 含
AL	① 在组合式 BCD 码的加减法校正指令中. ②在 XLAT 指令中，作目的寄存器.	隐 含 隐 含
BX	在 XLAT 指令中，作基址寄存器.	隐 含
CX	在循环指令中. 作循环次数计数器.	隐 含
CL	在移位指令中. 作移位次数计数器. (指令执行后，CL 中内容不变)	特定使用
DX	在字数据的乘法和除法指令中作辅助累加器. (即存放乘积和被除数的高十六位)	隐 含
SP	在堆栈操作(PUSH,POP,PUSHF,POPF) 中作堆栈指针.	隐 含
SI	在串操作指令中，作源变址寄存器 (MOVSB,MOVSW,LODSB,LODSW,CMPS)	隐 含
DI	在串操作指令中. 作目的变址寄存器. (MOVSB,MOVSW,STOSB, STOSW,SCAS,CMPS)	隐 含

二、段寄存器(Segment Register)

Intel 8086/8088 CPU 将存储器划分成若干段(如图 2.5 所示)，把将要运行的程序各部分(代码，数据……等)分别放在一个存储段中。每个存储段用一个段寄存器来指示它的首地址(即段基址)。当 CPU 访问某存储单元(如取指令或存取操作数)时，就必须指明该存储单元由哪个段寄存器指向，同时给出该存储单元在这个存储段内的偏移地址(即偏移量)。一个存储单元与它所在段的段基址之间的距离(以字节数计)叫该存储单元的偏移量。

一个程序划分成多少个存储段可以是任意的，用 CS,DS,ES,SS 段寄存器分别指明的段叫做当前段。在程序运行的任何时刻，最多只能有四个当前段。为了调换当前段，可以用程序的办法更换相应段寄存器的内容。例如某程序已设定两个数据段 DATA1,DATA2，当数据段寄存器 DS 指向 DATA1 段基址时，DATA1 为当前数据段。如现在程序要求访问数据段 DATA2 中某存储单元，这时必须先转换当前数据段，用几条指令让 DS 指向数据段 DATA2 的段基址，使 DATA2 成为当前数据段。

这四个段寄存器有各自的作用，不能互换。CS 一定是指向存放有指令代码的各个代码

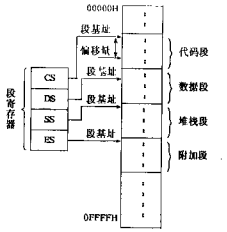


图2.5 用段寄存器寻找存储段

段，SS 是指向被开辟为堆栈区的各个堆栈段，DS 和 ES 通常是指向存放数据和工作单元的数据段。

三、指令指针 IP(Instruction Pointer)

IP 与其他计算机和微处理器中程序计数器 PC 的作用类同，它是指令的地址指针。在程序运行期间，CPU 自动修改 IP 的值，使它始终保持正在执行指令的下一条指令代码的起始字节的偏移量，如图 2.6 所示。

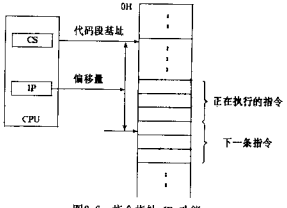


图2.6 指令指针 IP 功能

我们编制的程序不能直接访问 IP，即不能用指令去取出 IP 的值或给 IP 设置给定值。但是可以通过某些指令的执行而自动修改 IP 的内容。例如转移指令 JMP，JNE 等的执行，把目的地址的偏移量送入 IP；调用子程序指令 CALL 执行时，IP 原有内容自动压入堆栈，把子程序入口地址偏移量自动送入 IP。当从子程序返回主程序时，返回指令又自动从堆栈中弹回原有 IP 的内容送回 IP。

四、标志寄存器(Flags Register)

8086/8088 CPU 设置了一个十六位标志寄存器，用来反映微处理器在程序运行时的某些状态。寄存器中有 9 个标志位，其中 6 位标志位(CF,PF,AF,ZF,SF,OF)作为状态标志，记载了刚刚执行完算术运算或逻辑运算指令后的某些特征。另外 3 个标志位(TF,IF,DF)作为控制标志，对执行某些指令时起控制作用。图 2.7 中除指明控制标志位外，其余均为状态标志位。

(1)进位位 CF(Carry Flag)

当进行算术运算时，如最高位(对字操作是第 15 位，对字节操作是第 7 位)产生进位(加法类运算)或借位(减法类运算)，则 CF 置'1'，否则置'0'。CF 也可在移位类指令中使用，用它保存从最高位(左移时)或最低位(右移时)移出的代码(0 或 1)。

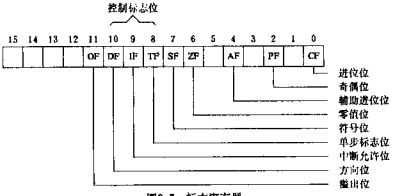


图2.7 标志寄存器

(2)奇偶位 PF(Parity Flag)

如操作结果低八位中含有'1'的个数为偶数时,则 PF 置'1',否则 PF 置'0'。PF 只检查操作结果的低八位,与该指令操作数的长度无关。

(3)辅助进位位 AF(Auxitiary Carry Flag)

当进行算术运算时,如低字节中低四位(即第 3 位)产生进位(加法)或借法(减法)时,则 AF 置'1',否则置'0'。AF 只反映运算结果的低八位,与操作数长度无关。它是用于十进制运算的调整。

(4)零值位 ZF(Zero Flag)

如运算结果各位全为'0'时,则 ZF 置'1',否则置'0'。

(5)符号位 SF(Sign Flag)

把运算结果视为带符号数。如运算结果为负数时,则 SF 置'1',为正数时,则置'0'。由于第 7 位/第 15 位是字节操作数/字操作数的符号位,所以 SF 与运算结果的最高位(第 7 位/第 15 位)相一致的。

(6)溢出位 OF(Overflow Flag)

当运算的结果超过机器用补码所能表示数的范围 N 时,则 OF 置'1',否则置'0'。对于字节运算:$-128 \leqslant N \leqslant +127$;对于字运算:$-32768 \leqslant N \leqslant +32767$。产生溢出一定的是同号数相加或异号的数相减。

溢出与进位是两个不同的概念,不能混淆。表 2.2 是以字节运算为例说明这两个不同性质的标志位。

(7)单步标志位(或跟踪位)TF(Trace Flag)

当 TF=1 时,在执行完一条指令后,产生单步中断,然后由单步中断处理程序把 TF 置'0'。TF 是供调试程序用。如在查错程序 DEBUG 中,可以使用单步命令,在每条指令执行后就停下来进行检查。

表 2.2 进位位与溢出位

加法举例			进位位 CF	溢出位 OF	结果
十进制	十六进制	二进制(补码)			
100D +) 100D 200D	64H +) 64H C8H	01100100B +) 01100100B 11001000B	0	1	出错
−85D +) − 1D −86D	ABH +) FFH [1] AAH	10101011B +) 11111111B [1] 10101010B	1	0	正确
− 86 +) −177 −202	ABH +) 8BH [1] 36H	10101011B +) 10001011B [1] 00110110B	1	1	出错

(8)中断允许位 IF(Interrupt—enable Flag)

当 IF=1 时，允许响应可屏蔽中断。相反，IF=0 时，不允许响应可屏蔽中断。

(9)方向位 DF(Direction Flag)

DF 为串操作指令规定增减方向。当 DF=0 时，串操作指令自动地使变址寄存器(SI 和/或 DI)递增(即串操作是由低地址到高地址)。当 DF=1 时，则自动地使变址寄存器递减(即串操作是由高地址到低地址)。

§2.4 存储器(Memory)

一、存储器的组成

存储器是由若干存储单元组成的。存储单元的多少表示了存储器的容量。每一个存储单元有相同的二进制位数。八位二进制称为一个字节(BYTE)。8086/8088CPU 具有 1 兆字节(1048576 个字节)的寻址能力，即可以在 1 兆字节的存储器中寻找出所需的一个存储单元。每一个存储单元(字节单元)有一个唯一的编号——地址。在机器内部这个地址是用二进制无符号数来表示的，1兆字节的地址范围是$\underbrace{00\cdots\cdots00}_{20\text{个}}$～$\underbrace{11\cdots\cdots11}_{20\text{个}}$，如图2.8所示。为了方便书写和编程，在源程序中常常是用五位十六进制数或用符号来表示一个单元的地址。

存储器虽然是由字节单元组成的，但是任何两个相邻字节就可构成一个字(WORD)(十六位二进制代码)。用地址值较小的哪个字节单元地址作为这个字单元的地址。一个字有十六位，其中低八位在较低字节中，而高八位在较高字节中。例如图 2.9(a)中 09235H 字节单元中有数据 56H，09236 字节单元中有 34H，09237H 字节单元中有 12H。如果把 09235H 和 09236H 两个相邻字节组成一个字，那么 09235H 字单元的内容为 3456H。如果把 09236H 和

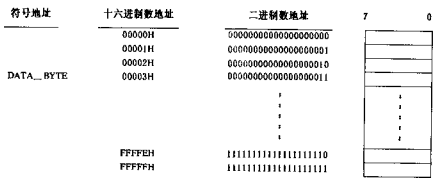


图 2.8 存储单元的地址表示

09237H 两个字节单元组成一个字，那么 09236H 字单元的内容为 1234H。在图 2.9(b)中，094A5H 字节单元中存放在字母 'B' 的 ASCII 码(42H)，094A6H 字节单元存放着字母 'A' 的 ASCII 码(41H)。如果把 094A7H 和 094A8H 两个字节单元组成一个字(字地址是 094A7H)，且要存放两个字母 'C' 和 'D' 的 ASCII 码，那么字 'C' 的 ASCII 码存放在高字节单元中，而字母 'D' 的 ASCII 码存放在低字节中。

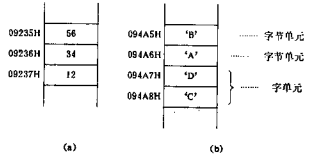


图2.9 存储器的字节和字

在 8086/8088 访问内存的指令中，分为字节访问和字访问两种，因此要求指令中表示出是哪种访问。在程序中，通常使用符号地址，在定义符号地址(变量名)时已指出它具有字节或字的类型属性，如在某指令中需临时修改某符号地址的字节或字的类型属性，可以选用适当的伪指令来说明(详见 § 4.2)。

二、存储器的段结构

8086/8088CPU 把 1MB 的存储空间划分成若干个段(Segment)，每个段至多由 64K(即 65536)个连续的字节单元组成，每个段是一个可独立寻址的逻辑单位。在 8086/8088 的程序中，需要设立几个段以及每个段的用途完全由用户自己确定。每个段中存储的代码或数据，可以存放在段内任意单元中。

每一个段在物理存储器(Physical memory)中有一个段的段基址(Segment Base Address)。每个段基址都是 16 的整数倍(即段基址的最低四位二进制数均为 '0')。各个逻辑段在物理存储器中可以是邻接的(Contiguous)，间隔的(Disjoint)，部分重叠的(Partly overlapped)和完全

重叠的(Full overlapped)等四种情况(如图 2.10)。所以一个物理存储单元可映象(Map)到一个或多个逻辑段(Logical Segment)中。例如图 2.10 中 DATA_BYTE 单元可映象到段 2,段 3 和段 4 中。

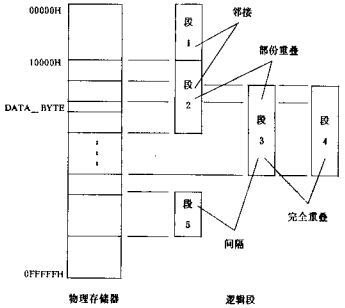


图2.10 物理存储器中的段结构

一个存储器虽然可以划分成若干个段,但是在任何时刻,一个程序只能访问四个段中的内容,这四个段分别叫代码段(Code Segment),数据段(Data Segment),堆栈段(Stack Segment)和附加段(Extra Segment)。它们分别由对应的四个段寄存器 CS,DS,SS,ES 指向。这四个段寄存器分别保存这四个段基址的高十六位二进制数——段基值(Segment Base Value)。由四个段寄存器指向的那些段叫当前段(Current Segment)。当前段至多可容纳 64KB 的代码,64KB 堆栈和 128KB 的数据(分别由 DS,ES 指向的当前段)。在规模不太大的应用程序中,上述这些容量是够用的。如果应用规模较大,我们可以在程序中通过修改相应段寄存器的内容,从而访问其他段。如图 2.11 中,由于四个段寄存器有段 B,F,I,K 的段基值,所以段 B、段 F、段 I、段 K 是当前段。如程序中需访问另外段(如段 J)中数据,那么可用程序办法,修改 DS 或 ES 的内容为 J 的段基值,改变当前段。

三、逻辑地址与物理地址

在 8086/8088 系列微机中,每个存储单元都有两种地址:物理地址(Physical Address)和逻辑地址(Logical Address)。在 1MB 的存储空间中,每一个存储单元(即一个字节单元)的物理地址是唯一的,就是这个单元的地址编码。物理地址由二进制的 20 位组成,它的范围是 00000H～0FFFFFH。CPU 与存储器之间的任何信息交换,都是使用物理地址。

在程序设计中,使用逻辑地址,而不使用物理地址,这不仅有利于程序的开发,且对存储器的动态管理也是有利的。一个逻辑地址是由段基值和偏移量(OFFSET)两部分组成,它们都是无符号十六位二进制数。段基值是一个段起始单元地址(段基址)的高十六位,它存放在某一个段寄存器中。偏移量表示了某存储单元与它所在段的段基址之间的字节距离。当偏

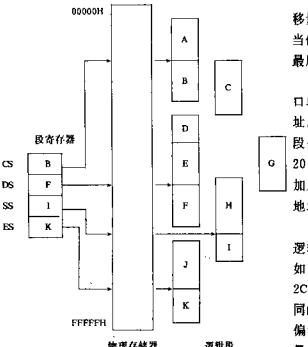


图2.11　存储器中当前段

移量为'0'时，就是这个段的起始单元。当偏移量为0FFFFH时，就是这个段的最后一个字节单元。

每当CPU访问存储器时，总线接口单元BIU把逻辑地址转换成物理地址。转换方法是：首先把逻辑地址中的段基值(在段寄存器中)左移4位，形成20位的段起始地址(段基址)，然后再加上16位的偏移量，产生20位的物理地址，转换过程如2.12所示。

由于逻辑段可以**重叠**，因此不同的逻辑地址可以得到同一个物理地址。例如图2.13中，有两个段基址2B0H，2C0H。某存储单元2D3H可以用两个不同的逻辑地址表达：①段基值为2BH，偏移量为23H；②段基值为2CH，偏移量为13H。但是经过逻辑地址到物理地址的转换，均转换为同一物理地址：2B0H + 23H = 2D3H，2C0H + 13H = 2D3H。这也说明：存储单元2D3H既可映象在段基值为2BH的逻辑段中，也可映象在段基值为2CH的逻辑段中。

在程序的执行过程中，CPU根据不同操作类型访问存储器，逻辑地址来源是不一样的。表2.3给出了不同操作类型，获得段基值和偏移量的不同来源。当在存储器中进行取指令操作时，段基值一定是由代码段寄存器CS提供，而偏移量从指令指针IP中获得。如进行堆栈操作，一定是从SS中获得当前堆栈段的段基值，堆栈指针SP含有堆栈顶部与堆栈段段基址之间的偏移量。对存储器中的变量进行存取操作时，多数都是在当前数据段中，即隐含着由DS提供段基值。如存取的变量在其他当前段中，允许用其他段寄存器(如CS，SS，ES)来提供该变量所在段的段基值，而变量的偏移量是由CPU根据指令中指定的寻址方式进行计算而得到的。根据寻址方式计算出来的偏移量又叫操作数的有效地址EA(Effective Address)。当使用串操作指令对串操作变量进行操作时，逻辑地址来源与一般变量操作略有不同。取源串通常是由DS提供当前数据段的段基值，也可以由其他段寄存器(如CS，SS，ES)提供段基值，而偏移量一定是由SI提供。对目的串操作只能由ES提供段基值，它的偏移量一定是由DI提供。当指令中用BP作基址寄存器来寻找操作数时，段基值通常由SS提供，当然也可以指定其他段寄存器，同一般变量操作一样，根据寻址方式计算出的有效地址EA作为它的偏移量。

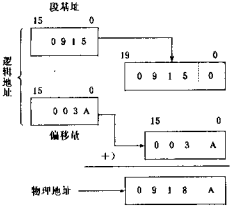


图2.12 逻辑地址到物理地址的转换

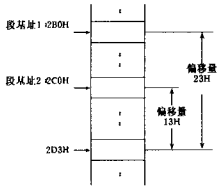


图2.13 存储单元的逻辑段映象

表2.3 逻辑地址的来源

序号	操作类型	逻辑地址		
		段基值		偏移量(OFFSET)
		隐含来源	允许替代来源	
1	取指令	CS	无	IP
2	堆栈操作	SS	无	SP
3	取源串	DS	CS,SS,ES	SI
4	存目的串	ES	无	DI
5	以BP作基址	SS	CS,DS,ES	有效地址EA
6	存取一般变量(除上述3,4,5项外)	DS	CS,SS,ES	有效地址EA

§2.5 堆栈(Stack)

堆栈是一个特定的存储区。在这个存储区中信息的进出严格按照一定规则进行。堆栈主要用于暂存数据和在"过程"调用或处理中断时暂存断点信息。不仅在现代程序设计中经常使用堆栈,在其他一些领域(如编译技术)中亦广泛的应用堆栈的概念。

一、堆栈的构造

现在通常是采用软件堆栈,由程序设计人员用软件在存储器中划出一块存储区作为堆栈(如图2.14)。这个存储区的一端是固定的,另一端是浮动的。所有信息的存取都在浮动的一端进行。这个存储区最大地址的存储单元为堆栈底部——叫栈底(BOTTOM)。在堆栈中存放的数据或断点信息从这里开始,逐渐向地址小的方向"堆积"。在任何时刻,存放最后一个数据的存储单元(即已存放数据的最小地址单元)为堆栈顶部,叫栈顶(TOP)。栈顶是随着存放数据的多少而变的,它是这个存储区的浮动"端头",而栈底是固定不变的,它是固定"端头"。

由于堆栈顶部是浮动的,为了指示现在堆栈中存放数据的位置,通常设置一个指针

——堆栈指针 SP(Stack Pointer),它始终指向堆栈的顶部。这样,堆栈中数据的进出都由 SP 来"指挥"。

在堆栈中存取数据的规则是"先进后出"FILO(First—In Last—Out)。就是说最先送入堆栈的数据(在堆栈底部),最后才能取出。相反地,最后送入堆栈的数据(在堆栈顶部),最先取出。这个存取数据的规则如同货栈一样,压在最底层的货(即栈底)只有待上面所有货物取走后才能取出。

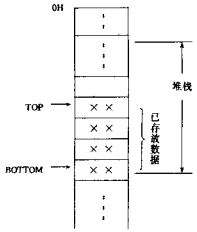


图2.14 堆栈构造

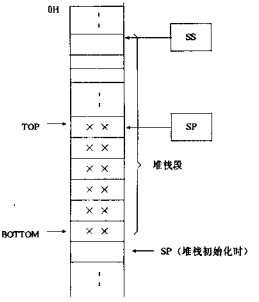


图2.15 8086/8088 堆栈构造

二、8086/8088 堆栈的组织

在 8086/8088 系列微机中,堆栈是由堆栈段寄存器(SS)指定的一段存储区(图 2.15)。通常,堆栈段中所包含的存储单元字节数就是堆栈深度(即堆栈的长度)。堆栈的底部是堆栈段的最大单元地址,堆栈顶部(栈顶)由堆栈指针 SP 指向。SP 中始终包含有段基址与栈顶之间的距离(字节数)。当 SP 初始化时,它的值就是这个堆栈的长度(这时它指向栈底+1 单元)。由于 SP 是十六位的寄存器,因此堆栈深度最大是 64K 个字节(一个段的长度)。

8086/8088 系列微机的堆栈是按字组织的,即每次在堆栈中存取数据均是两个字节,数据在堆栈中存放的格式是:

09154H	34	数据低八位
09155H	12	数据高八位

由于 8086/8088 系列微机中一个堆栈的长度是有限的,如果用户程序要扩大堆栈区域或者要更换一个堆栈区(已设置几个堆栈段),这时可以用重新设置堆栈段寄存器 SS 的办法来实现。在用户程序中每次更换堆栈段寄存器时,必须紧接着赋值 SP 的新值。