8088

汇编语言

IBMPC 丛书

5

PC 丛书编辑部

目　录

第一章

第二章

第三章

第四章

第五章

第六章

第一章 引 言

1981年8月，IBM介绍了它的个人计算机，绰号为“IBM PC”。这个PC把若干个重要的技术进展合为一体。其中最重要的是采用Intel8088微处理器作为其中央处理单元。

8088虽然在技术上是个8位芯片，但它支撑与它的哥哥16位8086同样的体系结构。这给PC以利用大数量存贮器的能力；大多数微计算机最多可以处理64K的存贮器，而PC却可以处理多达1024K或1兆字节。正是取这一特点的长处，IBM在本机的ROM（只读存贮器）里提供了大而强有力的BASIC语言解释程序。因此，写PC程序的大多数人用BASIC语言并不奇怪。

BASIC的问题

虽然BASIC语言易学易用，但有一定不足之处。BASIC程序设计者把PC视为一台有能力执行BASIC语句和功能的计算机。然而，实际上PC只有执行它的中央处理器8088提供的那些功能的能力。比起熟知的BASIC语句，这些机器指令原先的量就大得多。例如BASIC语句PRINT就是用100条8088机器指令完成的。执行BASIC程序时，轮到每条语句必须被译码。然后，使用相应的机器指令序列实现该语句的执行。图1—1中说明了这个过程，这个过程称为解释执行。当然这个过程本来就慢。

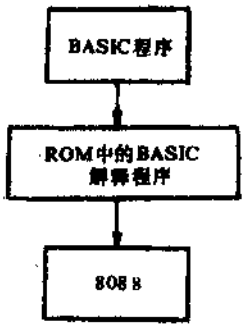


图1—1 解释执行

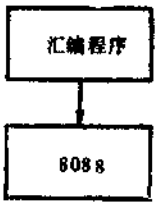


图1—2 汇编程序执行

如果我们写一个只由机器指令组成的程序，则可由8088中央处理器直接执行它。这种程序执行起来比它的BASIC对应程序快好多倍，因为我们省略了对每条程序语句必须做的译码辅助操作。图1—2说明这一点。机器语言组成的程序是用汇编语言写的。用来生成8088机器语言程序的汇编语言称为8088汇编语言

采用汇编语言所得的好处不仅是执行速度。如前所述，BASIC程序设计者把计算机视为BASIC计算机。这就限制他或她只能用BASIC语言提供的那些特点和手段。然而，汇编语言程序设计者把计算机看成它的最低层，并取用机内每一硬件特点的优点。亲自体验观看你自己的汇编语言程序在PC上建立、运行，是非常愉快的。你知道你是在控制使机器工作的那个电路。

能力与尽责

汇编语言的能力是难以驾驭的。但有一个与使用相关的代价问题（任何时候都不会有无代价的东西）。要编好汇编语言程序，你必须在本质上熟悉计算机的内部组分。其中最

重要的是8088微处理器。我们将研究它的内部结构并学习它能执行的许多指令。

在PC系统中还有其它我们必须好好研究的组分。例如，有一个芯片，它用来准确地计时事件。它是可编程的而且在我们的程序控制之下能用来完成各种定时功能。然而，要使其成为可能，我们必须首先懂得它的内部结构以及如何把它结合到PC的其它部分中。类似地，有专门的芯片和/或电路控制扬声器，处理中断，维护视频显示，等。如果我们要控制这些系统，必须懂得其中的每一个。

另外，我们必须知道系统支撑程序。这些程序是PC带来的，一些是永久地放在机器的ROM中而另一些则由DOS软盘片提供。这些程序建立软件环境，以允许我们写和运行我们自己的汇编语言程序。我们写的任一程序都必须在这个环境下运行，因此我们必须懂得它。

告　示

在后面章节里我们将包括上述所有问题。用这本书不要求先有汇编语言的经验。不过，读者应当熟悉写计算机程序时引用的一些概念，而且至少应熟悉一种程序设计语言，如BASIC。

第二章讨论了二进制和十六进制数制并介绍了汇编语言的结构。如果你熟悉其它汇编语言，你或许可以跳过这一章（特地告诫：很多公用汇编语言的约定不适用于8088）。

第三章叙述了8088的体系结构。这里涉及到寄存器，存贮器寻址方案，和非常强有力的指令集。在第四章，我们将学会如何使用IBM宏汇编程序。我们也从汇编语言程序设计者的观点详细地看一看IBM DOS。这一章包括了很多示范程序中的第一个。这些程序是可以实际地打入并运行在你的PC上。为此，你将需要至少64K的存贮器，一个磁盘驱动器，单色适配器和显示器，以及IBM DOS和IBM宏汇编程序。读者应当熟悉标准的DOS操作，比如建立和维持磁盘文件。应该懂得EDLIN程序，它是用于进入并修改程序文本的。

第五章叙述了PC系统板，它是个人计算机的心脏。我们将学会如何控制本机的中断结构以及键盘和定时机构。第六章包括了单色和彩色适配器相关的示范程序。当然，你会需要适配器和彩色显示器。

第七章说明串行通信背景的主要概念并叙述了如何使用异步通信适配器。与远程位置上的另一计算机通信的能力，开辟了一个完整领域的可能性。还介绍了PC转换成数据终端的示范程序。为运行这个程序，需要异步通信适配器。

最后，第八章包括了磁盘输入和输出。磁盘操作是在两个不同的级别上叙述的。高级的，允许我们操纵标准的DOS文件；而低级的，允许我们直接访问磁道和扇段。

第二章　基本概念

正如我们已经看到的，一个汇编语言程序设计员对他的计算机内部各部件必须有个本质的认识。在这一章，我们将研究所有现代微计算机的基本组成块块，而且对什么使这些机器能（时钟般地）持续工作有个较深的认识。

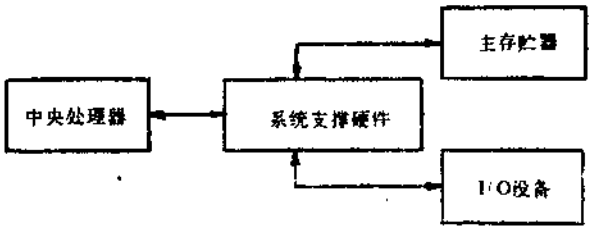


图2—1　微计算机的基本部件

把今天的微计算机逻辑地可分割成四个基本的部件，如图2—1.所示。该系统的心脏是**中央处理器**，它有能力取出并执行构成计算机程序的那些指令。**系统支撑硬件**为中央处理器的数据流进出提供通路，也实现保持中断轨迹，支撑用来维持计算机内当天时间的“心脏跳动”等诸如此类的各种任务。**主存贮器**是用于存贮由中央处理器直接访问的信息。I/O设备提供该系统和外界之间的接口。键盘和视频显示是最通用的I/O设备，而大容量数据存贮设备如磁盘和带驱动器也归于这个范筹。

信息的电气表示

在图2—1描述的微计算机系统能够十分迅速和准确地存贮和处理信息。为了给这个计算机编程序，我们必须知道在计算机内部怎样表示信息。在本系统中信息的基本单位是十进制数字，这些数字共十个，即0，1，2，3，4，5，6，7，8和9。在计算机里是以电气信号的存在与否来存贮信息的；因此只能是两个而不是十个不同的数字。我们把这些数字叫0和1。计算机内的所有信息都是按这两个不同数字的各种组合来存贮的。

因为有两个数字，所以我们说计算机数制是**二进制数制**，并且我们用述语**二进制位（bit—比特，这是对二进制数字的缩写词），它指的是信息存贮的基本单位，当然这个存贮是能承受值0或值1的。**

如果我们要表示其值不是0和1的数，显然，我们必须要用到一个以上的信息存贮单位或者比特。现在如果把两个比特放在一起，它们就能承受四个不同组合中的一种：两个均为0，两个均为1，头一个为0第二个为1，和头一个为1第二个为0。我们用这些不同的组合可以表示0到3的数。如果我们加上第三个比特，我们就会有八种不同的组合，就能表示0到7的数。很明显，每加一个比特，我们能表示的值的范围就加倍；这样，四个比特就能表示16个不同的值（图2—2）。

我们要用确定比特个数的方法来表示若干项信息，定义出有多少个可能的不同值来承担项。这些不同的值能表示的数的范围为从0到某一上限，正如我们早已看到的。然而，重要的是要认识到：这种方法不仅仅限于表示数。例如我们要存贮一个物体的颜色，它是四个可能的值红、绿、蓝、或者黄之一，那么我们就可以用两个比特，如图2—3所示。

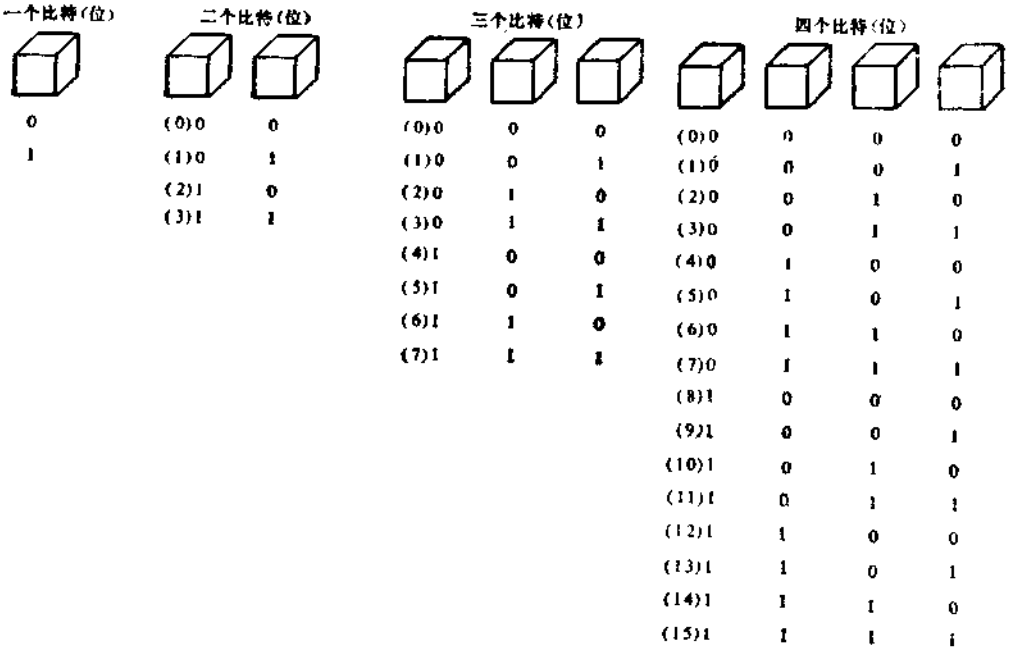


图 2—2　用比特表示数的数据

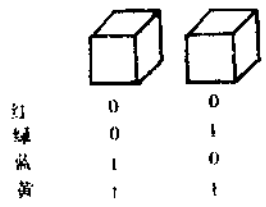


图 2—3　用两个比特表示四种不同颜色

这里，两个比特产生四种可能的值，其中每一个被指派给一个特定的颜色。

以前，曾明智地决定了：若使计算机成为实际有用的，我们必须能使用常规的单词和符号以及适当长度的数字串与计算机通信。为了做到这一点，需要有一种方法以表示字母表中的每一个字母以及所有的数字和所有标点符号。我们将要表示的，总共有100左右不同的字符。采用七个比特，我们能表示128个不同的值，如表 2—1 所示，而且容易处理上档和下档两个档的字母表的字符、数和标点符号以及各种“控制”码。这些字符的表示法是已经标准化了并且用于整个计算机工业中。这就是通常所说的美国信息交换标准代码（American Standard Code for Information Interchange）或简称ASCII。有这么个标准是个非常好的事，因为我们知道：实际上我们能走向任何计算机并且它会识别二进制码1000001为上档字母A。

如果给这些七位比特ASCII组再加1个比特，我们就有通常说的**字节**（byte），它是计算机内部存贮信息的最通用的基准单位。八比特字节能取256个不同的值，其头128个可解释为现有ASCII字符组部分。字节也可用来表示数 0 到255的值。当我们需要表示较大的数时，我们常常把两个字节配对而成所谓的**字**（Word）。字由16个比特组成，能取65,536个不同的值，而且常来表示从 0 到65,535 范围的数。在图 2—4 里给出了这些存贮信息的不同单位。注意：这些比特是从右到左地编号的，并从 0 开始。当我们需要访问一个字节或字中指定的某一比特时这个编号对我们是很方便的。

二进制数制是用来定义图2—4所给的组合中的每一个比特的数值。你大概熟悉我们自己的十进制数制，这里有十个不同的数字，而且每一数字位置代表10的不同乘方。这样，十进制数916由百位（10的2次方）上的数字9，加上十位（10的1次方）上的数字1，加上个位（10的0次方）上的数字6组成。在二进制数制中，只有两个不同的数字，而且每一数字位代表一个逐次的2的乘方。因此二进制数101就由4位（2的2次方）上的数字1，加上2位（2的1次方）上的数字0，加上个位（2的0次方）上的数字1组成。如图2—5所示，与二进制数101等价的十进制值为5。

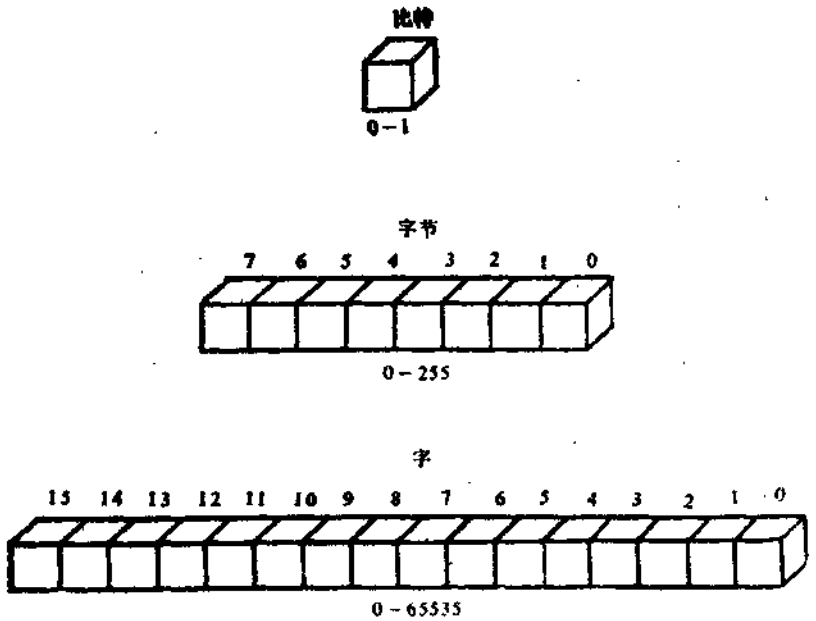


图2—4　存贮信息的单位

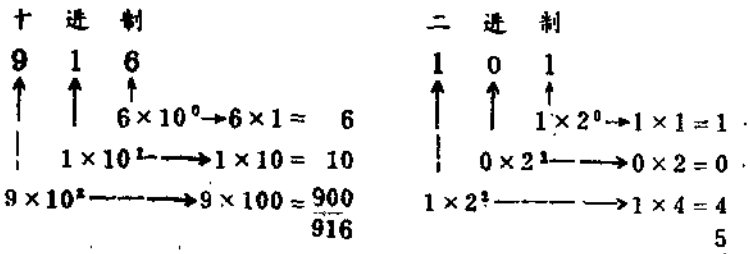


图2—5　十进制与二进数制

我们已经看到：当我们的数取得较大时，为表示该数所需要的比特个数增长很快。为了避免陷入长串1和0的困境，我们采用十六进制数制以作为一种速记。十六进制（或简称“hex”）是一个16为基的数制，有16个数字，如图2—6所示。注意：对十六进制的数字，我们用熟知的0到9作为其前十个数字。我们还需要六个，我们就用字母表中的头六个；这样十六进制数字A的值是10，数字B的值是11，以此类推最后的十六进制数字F的值是15。在十六进制里每一个数字位代表16的一次方。这样，这些数字位，从右到左地叫做

表2—1　美国信息交换标准代码（ASCII）

二进制 Binary	十六进制 Hex	字符 Char	二进制 Binary	十六进制 Hex	字符 Char	二进制 Binary	十六进制 HeX	字符 Char
0000000	00H		0100011	23H	#	1000110	46H	F
0000001	01H		0100100	24H	$	1000111	47H	G
0000010	02H		0100101	25H	%	1001000	48H	H
0000011	03H		0100110	26H	&	1001001	49H	I
0000100	04H		0100111	27H	'	1001010	4AH	J
0000101	05H		0101000	28H	(	1001011	4BH	K
0000110	06H		0101001	29H	)	1001100	4CH	L
0000111	07H	〈BELL〉	0101010	2AH	*	1001101	4DH	M
0001000	08H	〈BKSP〉	0101011	2BH	+	1001110	4EH	N
0001001	09H	〈TAB〉	0101100	2CH	,	1001111	4FH	O
0001010	0AH	〈LF〉	0101101	2DH	-	1010000	50H	P
0001011	0BH		0101110	2EH	.	1010001	51H	Q
0001100	0CH		0101111	2FH	/	1010010	52H	R
0001101	0DH	〈CR〉	0110000	30H	0	1010011	53H	S
0001110	0EH		0110001	31H	1	10 0100	54H	T
0001111	0FH		0110010	32H	2	1010101	55H	U
0010000	10H		0110011	33H	3	1010110	56H	V
0010001	11H		0110100	34H	4	1010111	57H	W
0010010	12H		0110101	35H	5	1011000	58H	X
0010011	13H		0110110	36H	6	1011001	59H	Y
0010100	14H		0110111	37H	7	1011010	5AH	Z
0010101	15H		0111000	38H	8	1011011	5BH	[
0010110	16H		0111001	39H	9	1011100	5CH	\
0010111	17H		0111010	3AH	:	1011101	5DH	]
0011000	18H		0111011	3BH	;	1011110	5EH	^
0011001	19H		0111100	3CH	<	1011111	5FH	—
0011010	1AH		0111101	3DH	=	1100000	60H	`
0011011	1BH		0111110	3EH	>	1100001	61H	a
0011100	1CH		0111111	3FH	?	1100010	62H	b
0011101	1DH		1000000	40H	@	1100011	63H	c
0011110	1EH		1000001	41H	A	1100100	64H	b
0011111	1FH		1000010	42H	B	1100101	65H	e
0100000	20H	space	1000011	43H	C	1100110	66H	f
0100001	21H	!	1000100	44H	D	1100111	67H	g
0100010	22H	"	1000101	45H	E	1101000	68H	h

续表1

二进制 Binary	十六进制 Hex	字符 Char	二进制 Binary	十六进制 Hex	字符 Char	二进制 Binary	十六进制 Hex	字符 Char
1101001	69H	i						
1101010	6AH	j						
1101011	6BH	k						
1101100	6CH	l						
1101101	6DH	m						
1101110	6EH	n						
1101111	6FH	o						
1110000	70H	p						
1110001	71H	q						
1110010	72H	r						
1110011	73H	s						
1110100	74H	t						
1110101	75H	u						
1110110	76H	v						
1110111	77H	w						
1111000	78H	x						
1111001	79H	y						
1111010	7AH	z						
1111011	7BH	{						
1111100	7CH	|						
1111101	7DH	}						
1111110	7EH	~						
1111111	7FH	△						

注：用〈 〉表示控制码（BKSP＝退格，TAB＝制表，LF＝换行，CR＝回车。）

十六进制数字	四个二进制数的等价值	十进制等价值
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

图2—6 十六进制数制

个位位（16的0次方），16位位（16的1次方），256位位（16的2次方），以此类推。我们将总是以字母H为后缀表示十六进制数。因此，当我们写数2FH时，我们表示的值等于2×16加上F（15）×1，就等于十进制的47。这就可以避免数与标号的混淆，因为标号总是以字母开始的。

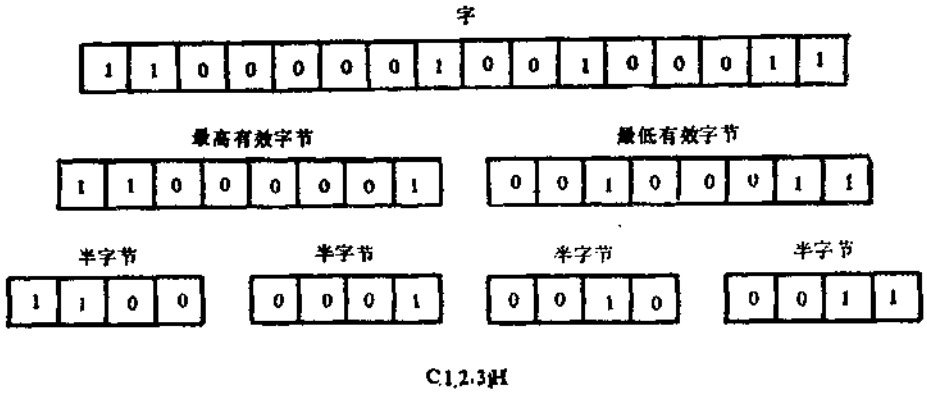


图2—7 最高有效和最低有效元素

现在我们能把我们的八比特字节分成高半个四比特和低半个四比特并用一个十六进制数字表示每半个字节。通常把字节写成：一对十六进制数字加上后缀H。十六进制数字，是半个字节，通常称为**半字节**（nibble）。这样，每一个**字节**包括一个最左边的或**最高有效**半字节以及一个最右边的或**最低有效**半字节。类似地，每一个字包括一个最左边的或**最高有效**字节以及最右边的或者**最低有效字节**。这后两个术语通常分别叫**MSB**和**LSB**（最高有效字节和最低有效字节）。图2—7中以图形定义了这些术语。

二进制和十六进制的运算

由于我们将以二进制和十六进制数制工作，我们必须明白如何用它们进行运算。这个过

程类似于十进制中用的我们都熟悉的过程。当我们加两个十进制数时，我们是从右到左地工作，加每列中的每一对数字。当一对数字之和超过9（最大的十进制数字）时，我们把1延送到下一个数字位并把减10后的余数放在当前的列里。在二进制中，只要其和超过1（最大的二进制数字），我们必须把1延送到下一个数字位并且将减2之后的余数放在当前的列中。类似地，在十六进制中，只要其和超过FH（15），我们必须把1延送到下一个数字位并把减16后的余数放在当前的列中。图2—8中给出这种操作的例子。

直到现在，我们用的仅仅是绝对值亦即正数。在二进制中要表示负数，需要用叫做二进制补码的方案。在一个二进制补码数里，最左边的（最高有效）位用来表示该数的符号。

二 进 制

```
  01101
+ 11001
```

从右到左算：1+1=2，进位1并置2－2=0；

```
    1
  01101
+ 11001
    0
```

1+0+0=1；

```
    1
  01101
+ 11001
   10
```

1+0+=1；

```
    1
  01101
+ 11001
  110
```

1+1=2，进位1并置2－2=0；

```
  1   1
  01101
+ 11001
  0110
```

1+0+1=2，进位1并置2－2=0

```
  01101
+ 11001
100110 = 结果
```

十 六 进 制

```
  5C3
+ 3D25
```

从右到左算：3+5=8；

```
  5C3
+ 3D25
    8
```

C+2=12+2=14=E；

```
  5C3
+ 3D25
  E8
```

5+D=5+13=18，进位1并置18－16=2；

```
1
  5C3
+ 3D25
 2E8
```

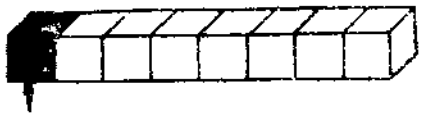
1+3=4

```
1
  5C3
+ 3D25
42E8 = 结果
```

图2—8 二进制和十六进制的加法

如果这个符号位是0，则该数为正数。如果该符号位为1，则该数为负数。要得到任一个二进制补码数的否定，我们简单地把所有比特取反（0变1和1变0）然后加1。在图2—9中说明了这一点。

用这种方案，我们能表达的最大正数和最大负数依赖于我们有多少比特。对于8比特的字节，能表达的最大正数是01111111，或7FH，或+127。最大负数是10000000，或80



举例

00000011

符号位＝0：正数＋3

11111100

符号位＝1：负数

要得等价正值

步1 翻所有位

00000011

步2加1

00000011

\+ 1

00000100＝4

因此11111100＝－4

图 2—9 二进制补码数

H，或－128（最小的负数将是11111111，或FFH，或－1）。对于16比特的字，最大正数将是0111111111111111的或7FFFH，或＋32,767。最大的负数将是1000000000000000，或8000H，或－32768。最小的负数将是1111111111111111，或FFFFH，或－1。然而，采用二进制补码表示法的重要着眼点是，我们以标准二进制算法能进行正数和负数一起的任意组合的加法，并能得到正确的结果。

主存贮器中信息的存贮

我们已经看到如何用比特串表示各种不同类型的信息。在我们的计算机用这些信息进行工作期间，计算机把这些比特存贮在**主存**中。主存是由大量的单元作成的，每一个单元能保持一个字节的信息，而你将召回的仍是等价的8位比特。给这些单元的每一个分派一个唯一的**地址**，从数0开始，对于连续的每一个单元递增1。中央处理器用指定该单元地址的办法存取包括在任一单元中的信息（图2—10）。存贮器地址通常是由中央处理器操纵的，就像操纵它含有的数据一样，而且通常用十六进制格式表示。把我们的计算机存贮单元的个数称为它的**存贮器大小**。这个数通常是超过“K”的好多倍，例如“32K”。“K”表示值1024，或400H。这样，一个32K的计算机就有32 • 1024，或32,768（8000H）个存贮单元。

在图2—10中我们看出存放在存贮器中的一个数据字节是如何用相应的地址访问的。要把一个字存放到存贮器中，就得用两个相邻的存贮器单元（看图2—11）。然后，访问这个字要用两个地址中的较低的地址。另外，组成一个字的两个字节是从我们希望的地址开始颠倒过来存放。把最低有效（最右边的）字节存放在低地址中，把最高有效（最左边的）字节

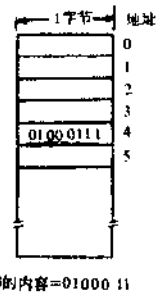


图 2—10　主存贮器

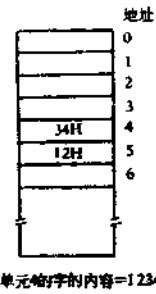


图2—11　主存里存放一个字

存放在较高的地址中。

中央处理器的作用

中央处理器的任务是执行存放在计算机主存里的指令序列。它给出一个**起动地址**，并开始从存贮器单元中取数据字节。中央处理器把每一个数据字节解释成为处理器能实现的一些功能的描述。跟在这个**操作码**字节的附加字节可以包括用来完成该功能的指定的数据。把这些附加的字节称为**操作数**。

处理器一旦完成了相应的功能，它继续取存贮器中的下一个相邻的操作码字节并执行其中规定的功能。由此可见，计算机的程序设计包括准备一系列有序字节，它们代表所要求的功能（并按所要的操作次序排列），并把这些数据字节存放到存贮器中，然后指示处理器在相应的地址上开始执行。

中央处理器能执行的功能的完整清单叫做它的**指令集**。指令集中的每一条指令是由不同的数据字节值来表示的。为完整地定义一条指令，需要一个以上的数据字节的情况是太多了。这种情况需要一个或更多的操作数字节。

中央处理器能执行的功能类型通常包括在系统内部把一个数据从一点挪到另一点，或者在处理器本身内部操纵数据。在处理器里有一组**寄存器**。当处理器正在处理数据时，这些寄存器用来保持这些数据。因此，一条典型的指令是要从指定的存贮器地址取出一个数据字节，并把它放到处理器内的特定寄存器里。下一条指令可以是把这个寄存器的内容加到另一个寄存器的内容。用这种方式，指令跟着指令地把大而复杂的计算编成程序。

我们为什么需要汇编程序语言

现今，微计算机的中央处理器通常是整个地包括在一个芯片上，并称为**微处理器**。尽管它的尺寸微小，微处理器有能力识别并执行上百条不同的指令。正如我们所看到的，在计算机内部每一条指令是以不同的比特模式来表示的。当我们编制这些指令序列时，我们不能为记住每条指令的本来比特模式而烦恼。换言之，我们将要用一个符号名字，或**助记符**来表示处理器能实现的各种功能的每一个。我们现在能用这种符号语言的命令序列来编写我们机器的程序。**汇编程序**把我们的符号程序转换成处理器能识别的实际比特模型，使得我们能把

这个程序装入到我们的机器里并执行它。

让我们看一下汇编语言程序的一个简单例子。这个程序的目的是要计算已放在存贮器里的两个数的和，减第三个数，并且把这一结果放回到某一单元等。

```
1    DATA      SEGMENT
2    INPUT1    DW  10
3    INPUT2    DW  25
4    INPUT3    DW  8
5    RESULT    DW  ?
6    DATA      ENDS
7    CODE      SEGMENT
8              MOV  AX, INPUT1  ; 得第一个数
9              MOV  BX, INPUT2  ; 得第二个数
10             ADD  AX, BX      ; 都加起来
11             MOV  BX, INPUT3  ; 第得三个数
12             SUB  AX, BX      ; 从第一个结果减去
13             MOV  RESULT, AX  ; 存放最终答案
14   CODE      ENDS
```

本程序的语句1告诉汇编程序，我们就要定义要放在存贮器的一个段里的信息，而且这个段将被叫做DATA。存贮器段的概念，将在稍后作说明，现在并不重要。语句2，3，4，和5中的命令DW告诉汇编程序在存贮器中要定义一个字，这个字包括跟在后面的那个值。在语句5，这个值被指定为问号，它告诉汇编程序：对于存贮器的这个字没有初值。在各自的语句上作为标号出现的INPUT1，INPUT2，INPUT3，和RESULT等诸名字，在本程序的后面被用来符号地指向马上要定义的存贮器地址。语句6的命令ENDS代替“段结束”并告诉汇编程序：我们现在结束DATA段的定义。在语句7里我们开始定义CODE段，它将包括我们程序的实际指令。语句8的命令MOV代替“传送”并告诉汇编程序：我们要把以标号INPUT1定义的存贮器的内容传送到AX为名的寄存器中。类似地，语句9要求把标号INPUT2的数据传送到寄存器BX。语句10的命令ADD将AX寄存器内容加到BX寄存器的内容并把结果返回给AX寄存器。照此下去，我们看到：语句11把INPUT3的数据传到BX寄存器，语句12从AX寄存器的内容减去BX寄存器内容。语句13指出把AX寄存器的内容传送到标号为RESULT的存贮单元中。最后，语句14指出汇编程序：已到了CODE段的结束。

从这一简单例子我们能学到很多有关汇编语言结构方面的东西。例如，我们看到：汇编语言程序的每一个语句是如何可分成四个独立的字段。如果有的话，第一个总是标号字段。它是用来给一个语句指派一个符号名字，如上面程序中的语句1—7和14一样。

标号字段后面是操作码字段。如果没用标号，那么在操作码之前必须有1个或1个以上的空格。操作码指定我们希望由汇编程序实现的功能。例如，在语句2中操作码“DW”告诉汇编程序：我们希望“定义一个字”。语句8的操作码“MOV”指示汇编程序要生成一条“传送”指令。

如果需要，则操作码字段后面是操作数字段。这个字段用来使要完成的功能更完善。例如，语句2中的操作数“10”告诉汇编程序：已经定义了要放到该字里的值是多少。通常，一条指令需要一个以上的操作数。在这种情况，操作数彼此间以逗号隔开。对要求两个操作数的多数指令而言，可以把其中之一看作是源而另一个看作是目的。标准汇编程序语言习惯上总是把目的操作数放在源操作数之前。

注解字段选择地跟在操作数字段后面。它总是以分号（；）开头。在任一汇编语言程序中，用许多说明性注解时分号显得十分重要。如果没有它，你只好找自己去读一个月前你自己写的程序并会问道：“这事是怎么做的？”。

然而，为了达到这一点，我们首先必须学习有关8088微处理器的大量知识。

第三章　8088的体系结构

在这一章，我们将看一看8088芯片的内部组分和存贮器的寻址方法。我们将学习8088的指令集并学习如何用它。这些概念构成8088的**体系结构**。

从图3—1中看出8088**CPU**（中央处理器）的框图。CPU和主存之间的数据传送是经过**存贮器接口**的，进来的数据，它是指定解释成指令的，是放到**指令串字节队列**中。**执行部件控制系统**从这个队列中取出指令并把它送到要解释和执行它的**执行部件**中。正在做这件事时，**总线接口部件**试图再装满指令串字节队列。这种设计被称为**流水线**，而且尽量保证每次CPU一旦结束正在执行的指令，就有另一条指令等待执行。大多数微处理器，执行一条指令之后必须等到从存贮器读出下一条指令为止。因为8088有流水线，所以能更快地执行指令。

执行部件里包括了**算术/逻辑部件**，或叫ALU。这个组分担负着所有算术运算和比较。为了产生一个数字结果，ALU经常置各种**标志**来指示结果的各种状态。例如，一个计算结果为—5，则**符号标志**应被置成能指示该结果是负的。

8088的寄存器组

图3—2中给出了8088的寄存器组。从程序设计者的观点看，在微处理器里这些寄存器是最重要的组分。当我们开发自己的汇编语言程序时，我们会经常使用和引用8088的各种寄存器。

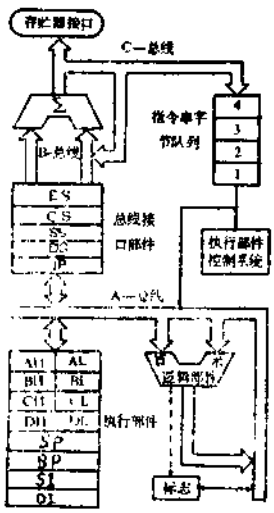


图3—1　8088的框图

有四个通用寄存器，每一个都是16比特宽，叫做AX，BX，CX，和DX。这四个中的每一个都能作为一对8比特（字节）寄存器被访问。这一对的左边（最高有效）字节为“高”字节，而右边（最低有效）字节为“低”字节。这样，寄存器AX可以被分为寄存器AH(“A高”)和AL（“A低”）；寄存器BX也可分为BH和BL；等等。

有两个**串变址寄存器**，都是16比特宽度。分别叫SI（“源变址”）和DI（“目的变址”）并通常用来指出存贮器中的数据串。它们也能作为通用16比特寄存器。

SP是**堆栈指针寄存器**，用来实现存贮器里的硬件堆栈（将会有堆栈用法的简短说明）。SP由**基址指针寄存器**BP补充，而BP也可作为一个16比特通用寄存器。

四个**段寄存器**是专用寄存器。每当8088访问存贮器时总是用到段寄存器。因此，每当执行任一指令时，微处理器总是隐含地访问一个或多个段寄存器。

另一个专用寄存器是**指令指针寄存器** IP，它指向要执行的下一条指令。最后是**标志寄存器**，它包括说明微处理器当前状态的若干个一位标志。