绪　论

一、Intel 微处理器发展史

1971 年，Intel 公司的第一块微处理器诞生了，它叫做 4004，是一块 4 位的 CPU。4004 很快被增强为 8 位 CPU——8008。在今天看来，4004 和 8008 这兄弟俩可能很不起眼，但在当时，它们确是了不起的新事物。它们曾在教学、程控、计算等应用领域中发挥过作用，然而并没有将它们正式装过象样的计算机。我们将这两种芯片叫做 Intel 公司的第一代微处理器。

1974 年，Intel 推出了它的第二代微处理器 8080。这是第一个通用的微处理器，它的出现，在微处理器工业史上是个十分重要的里程碑。8080 是 8 位的 CPU 芯片。1978 年推出的第三代微处理器 8086，标志着微处理器作为“实际的”计算机的开始。86 系列从此诞生了，8086 是该系列的第一个成员。8086 是 16 位的处理器。8086 的小兄弟 8088(准 16 位的，数据总线仍为 8 位)，曾经引起了个人计算机的革命。自从 1981 年 IBM 公司推出风靡全球的 PC 机以来，8088 微处理器结构几乎出现在每一台早期的 PC 及其兼容机上。不仅如此，由于 PC 及其兼容机的普及，和 PC 拥有的软件之丰富，众多其它类型计算机设备的设计者们也都乐意使用 8086 或 8088。这样一来，8086 结构从它诞生之日起直到现在，一直是世界上最主要的微处理器结构。

1982 年，86 系列增加了新成员 80186。该芯片在结构上与 8086 一样，只是在同一片芯片上增加了一些公用的系统。同年，8086 体系结构的超集 80286 问世。80286 能做 8086 所做的一切，并增加了许多功能，特别是对多任务的支持，也就是说，可以同时执行一个以上的应用程序或任务。为了支持多任务，80286 解决了任务或任务之间的，以及任务的内存空间之间的保护问题。80186 和 80286 都是 16 位的。86 系列中最年轻的也是功能最强的成员是 80386，它是 1985 年发布面市的。80386 比它的兄长们有两个最主要的增强：一是 32 位的操作和数据类型；二是采用了页式管理技术。在 80386 以前，86 系列的成员在内存管理上都只有用段式管理，而 80386 不仅采用段式管理，还采用页式管理。这，我们在第 4、5、6 章中还将详细讨论。除此之外，80386 还拓展了 80286 的多任务管理能力，可以同时执行 8086、80286、80386 的应用程序、任务，甚至操作系统。

最后值得一提的是浮点协处理器。86 系列的每一代产品都有其相应的浮点运算协处理器，如 8087 对应 8086，80287 对应 80286，80387 对应 80386。这些协处理器和它们的处理器紧密耦合，形成了能够支持浮点运算及其数据结构的计算机结构。

二、和 8086、80286 的兼容

86 系列的每一代新产品都和老的产品保持完全兼容。80386 可以执行 80286、8086 的任何程序(第 7 章将详细讨论)。但本书并不在兼容性上多费笔墨，而集中讨论 80386 上的全部 32 位机的特性，以及如何通过软件使 80386 成为一台实际的 32 位机。

本书对象为具备 8086 和 80286 编程经验的、从事计算机开发和应用的教育、研究人员和工程技术人员。所以在介绍它们兼容性的时候，只须指出系列产品和 80386 之间的区别就够了(请阅读附录 B)。如果读者还想了解 8086 和 80286 的编程的话，那么可以利用的资料和手段是相当丰富的，但它们不在本书的范畴之中。

第1章 数据类型

1.1 80386的数据类型

从本质上说，计算机所能做的全部工作可归结为对数据的存储、检索和处理。所以，想要掌握某种计算机的编程，必须先了解该机支持的数据类型。常用的数据类型有：带符号整数，无符号整数，BCD(包括压缩的和非压缩的二-十进制码)，字符串，位，浮点数。上述数据类型，大多数可在常见的机器中找到。这里，我们着重介绍80386和其它机器，尤其是和8086、80286在数据类型上的差别。

1.1.1 内存

在详细讨论数据类型之前，先说一说内存的组织。和所有的传统的计算机一样，内存是一切信息最主要的源泉和归宿。简单说来，内存即是有唯一地址的字节的有序阵列。在正常情况下，地址从0开始，一直排到机器所能支持的最高地址。80386是32位机，一共有2^{32}个物理地址，或者说成"总共有4G(4×10^9)字节的物理内存"。注意这里是"物理"地址范围。在第2、4、6章中，你可以看到：在段式管理和页式管理中的虚拟内存的地址范围，可以大大超过2^{32}(4G)。

某些类型的数据，其二进制的表达形式超出8位(16位、32位、甚至更多)，于是不得不用多个相邻字节来存放一个数据值。常见的有两个相邻字节组成的字(Word)和4个相邻字节组成的双字(DWord或Double Word)。对字来说，这个16位的数据就有2^{16}个可能的值。对双字来说，这是个32位的数据，就会有2^{32}种可能的值。

存取多字节数据，首先要解决的问题是：数据的最低有效8位是放在相邻字节(地址号连续)的地址最低者(地址号最小的字节)之中，还是放在相邻字节的地址最高者(地址号最大的字节)之中，也就是首先要解决字和双字存取的基准的问题。关于基准的方案不外乎有两种，请看图1.1和1.2。

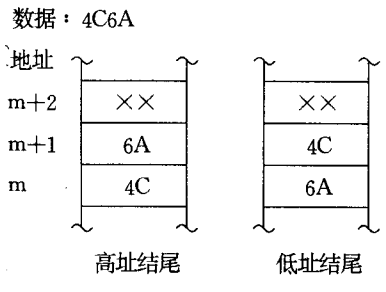


图1.1 高址结尾和低址结尾

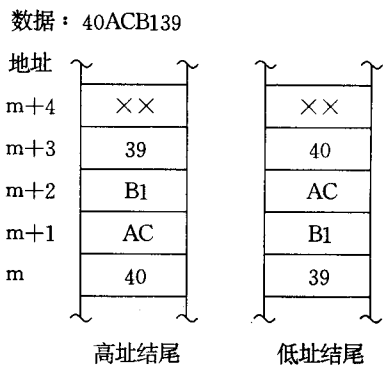


图1.2 双字的高址结尾和低址结尾

图 1.1 表示两个相邻字节存放一个字的方案。两图中左边的方案，将最低有效 8 位放在最高地址字节中，将最高有效 8 位放在最低地址字节中，这种方案被称之为“高址结尾(Big Endiun)”。两图中右边的方案，与左边的正好相反，将最低有效 8 位放在最低地址字节中，而将最高有效 8 位放在最高地址字节中，这种方案叫做“低址结尾(Little Endiun)”。

请读者记住，80386 采用的是低址结尾，即两图中右边的方案。若采用最低地址为(m)的两个相邻字节来存放一个字，该字的地址亦取(m)，该字的最低有效 8 位放在地址为(m)的字节中。

同理，若采用最低地址为(m)的 4 个相邻字节来存放一个双字的话，该双字的地址就是(m)，该双字的最低有效 8 位放在地址为(m)的字节中，而最高有效 8 位则放在地址为(m+3)的字节中。

1.1.2 约定

在这一节里，我们交代一下本书采用的有关数字表示方式的某些约定：

在数字后面加上“h”的，表示一个十六进制数。如 100h=256。在数字后面加上“b”的，表示一个二进制数。如 100b=4。在数字后面既没有“h”，也没有“b”，则表示一个十进制数。附录 D 给出了二进制、十六进制和十进制之间的转换表。

1.1.3 整数

和许多计算机一样，80386 也支持 8 位长度的字节、16 位长度的字、32 位长度的双字的整数。32 位的整数在 86 系列的其它机器 (8086、80186、80286)中是没有的。附录 E 给出了从 2^0 到 2^{32}的 2 的幂的表。

表 1.1 定义了全书通用的几种简略表示法和它们真正的值。

表 1.1 2 的次幂的缩简表

简 写	2 的次幂	十进制值
1K	2^{10}	1024
4K	2^{12}	4096
16K	2^{14}	16384
32K	2^{15}	32768
64K	2^{16}	65536
2G	2^{31}	2147483648
4G	2^{32}	4294967296

整数又分为无符号整数和带符号整数。80386 各种类型的整数和它们的有效数值范围如下页表：

长度	类　型	范　围	
		十六进制	十进制
8	字节的带符号整数 字节的无符号整数	80—00—7F 00—FF	−128 到 127 0 到 256
16	字的带符号整数 字的无符号整数	8000—0000—7FFF 0000—FFFF	−32K 到(32K−1) 0 到 64K
32	双字的带符号整数 双字的无符号整数	80000000—00000000—7FFFFFFF 00000000—FFFFFFFF	−2G 到(2G−1) 0 到 4G

1.1.3.1　无符号数整数

图 1.3 表示 80386 内存中无符号数据格式的位结构。

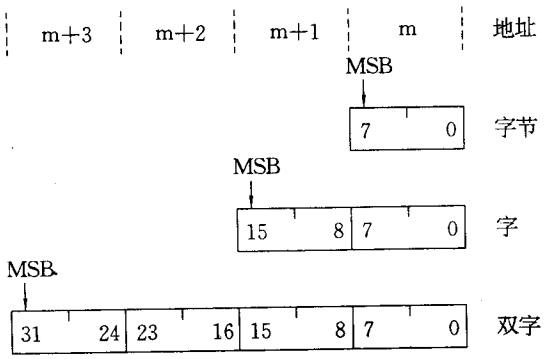


图 1.3　无符号整数

图中假定它们的起始地址均为 m，位 0 为最低有效位(LSB)，MSB 为最高有效位。

1.1.3.2　带符号整数

表示带符号整数的方法有以下四种：偏移值(Biased Numbers)，带符号的值(Sign Magnitude)，二进制反码(One's Complement)，二进制补码(Two's Complement)。

表 1.2 表示了 8 位长的带符号数的四种表示方法。80386 和 86 系列的所有成员都采用二进制补码作为带符号整数的表示法。

在描述二进制补码表示法之前，我们先简要介绍其余三种整数表示法，因为下面讨论浮点数据类型时，会用到这些表示法。

(1)　偏移值

由于偏移值表示法具有便于进行数据比较的优点，所以常用来表示浮点数的指数。在原来的正数或负数之上再加一个偏移量(Bias)，便得该数之偏移值。这个偏移量通常使表示法允许的最小负数变为 0，而使最大的正数变成表示法中的最大数。请看表 1.2 中的例子，其偏移量选择为 127，它使得最小负数(−127)的偏移值变成了 0，而最大的正数(128)则变成了 255。

(2)　带符号的值

所谓带符号的值即为数前冠以符号的值(或绝对值)。在带符号的数值的表示法中，有一

位用来表示符号(0 为正,1 为负),而其余的位则为值的二进制表示形式。浮点数的有效数字便是用这种方法表示的,其符号位给出了整个浮点数的符号。

表 1.2　带符号整数格式

十进制值	2 的补码	2 的反码	偏移值(=127)	带符号的值
128	无此值	无此值	11111111b	无此值
127	01111111b	01111111b	11111110b	01111111b
126	01111110b	01111110b	11111101b	01111110b
⋮	⋮	⋮	⋮	⋮
2	00000010b	00000010b	10000001b	00000010b
1	00000001b	00000001b	10000000b	00000001b
0	00000000b	00000000b	01111111b	00000000b
−0	无此值	11111111b	无此值	10000000b
−1	11111111b	11111110b	01111110b	10000001b
−2	11111110b	11111101b	01111101b	10000010b
⋮	⋮	⋮	⋮	⋮
−126	10000010b	10000001b	00000001b	11111110b
−127	10000001b	10000000b	00000000b	11111111b
−128	10000000b	无此值	无此值	无此值

(3)　二进制反码

对于正整数，其二进制反码与带符号的值具有相同的形式，即最高有效位为 0(表示正数),其余各位为该值(绝对值)的二进制表示。而负整数的二进制反码，由该数绝对值的二进制码逐位取反得到。由于此法简单,容易计算,所以在早期的计算机中用得很普遍，但现在用得少了。

(4)　二进制补码

所谓二进制补码,即是在二进制反码之上再加以 1。其最高有效位(MSB)也是符号位,而且也是 0 为正,1 为负。由于二进制补码易于实现,通过用于无符号数的简单的加法器就能实现从负数到二进制补码的转换,所以它得到了普遍的采用。这种优点对于既支持无符号数,又支持二进制补码数据的 80386 来说,也是颇为重要的。

图 1.4 给出了 80386 的二进制补码格式。

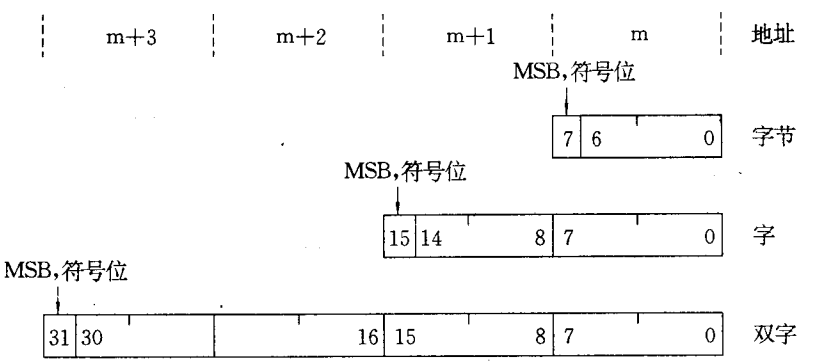


图 1.4　二进制补码字节、字、双字

80386 可对二进制补码进行各种各样的算术运算。第 3 章将讨论这些操作。和所有 86 系

列成员一样，80386 的二进制补码运算也能产生溢出。这些情况将在第 2 章中说明。

1.1.4 字符串

字符串就是字节、字、双字的相邻序列。对双字字串的支持只有 80386 才有，86 系列的其它成员是没有的。串的长度可从 1 到 2^{32} 个元素。图 1.5 显示了这三种类型的字符串。

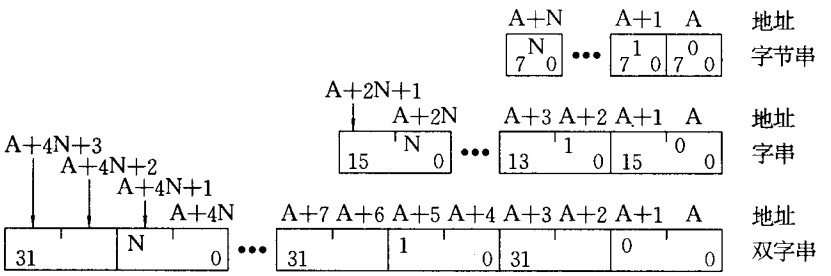


图 1.5 字节、字、双字字符串

80386 对字符串操作的指令有：将字符串从内存的一个区传送到另一个区、比较两个字符串、将一字符串用固定的值来填充、对 I/O 口的读写操作、按指定的数据值搜索字符串等。

1.1.5 ASCII

字符串的最常见的格式是 ASCII 码，因为大多数来自终端的数据都为 ASCII 码。附录 F 就是一个完整的 ASCII 码表。它包括数字、字母、特殊字符和控制字符。

在第 3 章中，我们将会看到 80386 支持对于 ASCII 码的诸如加和除的算术运算。

1.1.6 位

对位串操作的支持，在 86 系列中只有 80386 才有。

位支持是十分重要的，因为数据用单一的位来表示也是常见的。比如说，“位图”便是大家十分熟悉的例子。在位图显示中，每一个映射到内存中单一的位的“象素”，都用一个圆点(.)来表示。若某个圆点发光，表示该位为 1；若圆点不发光，表示该位为 0。还可以举些例子，如用位值来表示信号灯的忙(1)和闲(0)。当然，将一个完整的字节当作一个“象素”或“信号灯”不是不可以，但这样要浪费许多空间。仔细算一下，如果用一个字节，来表示上面例子中的一个数据元素，那么这个字节中有七位是无用的，也就是说，87.5%被浪费了。

80386 支持的位串，可以包含 2^{32} 个位，并通过一个带符号的双字来检索。到了第 3 章我们再谈它们的实际操作。图 1.6 表示一个位串在内存中的布局。位的索引是个带符号数。在位串中不存在最低有效位的概念；即使是第 0 位，也不是最低有效位。

一个位在位串内部的地址用 32 位的带符号整数来表示，这整数叫做“位偏移量”，取值范围从(−2G)到(2G−1)。位偏移量由字节地址和位余数组成。位偏移量除以 8，便得到该位所在字节的地址，而位偏移量对于 8 的模指出了被检索的位在所在字节中的位置。图 1.7 举了两个例子，图中两个位的位偏移量分别为(23)和(−18)，它们都在地址为 N 的位串中。在第 6 章我们还要详细研究。

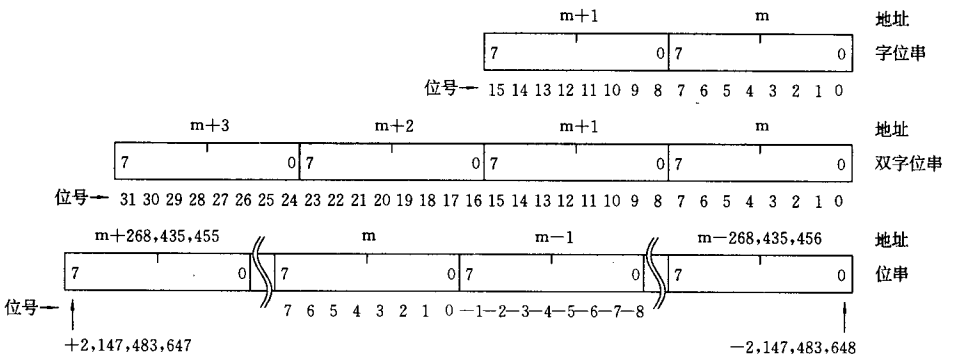
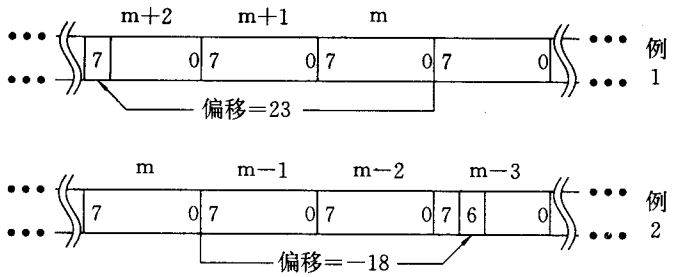


图 1.6　位数据类型



	例1		例2	
	字节索引	位余数	字节索引	位余数
	23	23	−18	−18
	÷ 8	mod 8	÷ 8	mod 8
	2	7	−3	6

图 1.7　位串偏移量的例子

1.1.7　BCD 码

在 8086、80286 或其它机器上编过程序的读者，都已熟悉了 BCD 码，所以用不着多介绍了。表 1.3 给出了一张简单的 BCD 编码表。还告诉大家，在第 3 章中有 80386 对 BCD 进行加减操作的指令。

表 1.3　十进制数的二进制编码

二进制数	十进制数	二进制数	十进制数
0000 0000b	00	0000 1010b	0X
0000 0001b	01	0000 1011b	0X
0000 0010b	02	0000 1100b	0X
0000 0011b	03	0000 1101b	0X
0000 0100b	04	0000 1110b	0X
0000 0101b	05	0000 1111b	0X
0000 0110b	06	0001 0000b	10
0000 0111b	07	0001 0001b	11
0000 1000b	08	⋮	
0000 1001b	09		

注意：对于 BCD 表示法 X 表示非法值

80386 只会直接处理单字节的 BCD 码。多字节的 BCD 码如何处理在第 3 章里讲。每字节放两个 BCD 码位，0 到 3 位放的是低位，4 到 7 位放的是高位。非压缩的 BCD 码只在每个字节的 0 到 3 位放置一个码位(4 到 7 位空闲)。

1.2 80387 的浮点数据格式

80387 是 80386 的协处理器，它和 8087、80287 十分类似。

1.2.1 浮点概述

到目前为止，我们讨论过的众多数据类型，无论是带符号整数、无符号整数、字符串、位串、ASCII和BCD都无法表示带有分数或小数的数。尤其是一些重要的物理常数，象：3.1416，8.854×10^{-14}，6.02×10^{23}，9.81，9.11×10^{-31}，1.38×10^{-23}等等，它们既有整数部分又有小数部分，在数学上称之为实数或有理数。在本书里，为了简单起见，统称它们为实数，不管这样称呼在数学上究竟是否严格。其实前面讨论过的整数也是实数，只不过是实数集合中小数部分为 0 的子集而已。

在计算机里，用来表示数的位数是有限的，所以计算机不能精确地表示出所有的实数。而只能表示无穷个实数中的一个极小的子集。然而，就是这个子集，已经可以解决极大多数的实际问题，而且丢掉的精度也是微不足道的。所谓“浮点数”，就是实数类型数据的计算机表示法。这里向读者强调两个不同的概念：“实数”是描写一个数的集合的数学术语；而“浮点”则是计算机用来表示实数的一个子集的数据类型。

下面我们来比较几种方法，以说明浮点运算的必要。假如现在让计算机来处理一笔款项，¥23.41。当然你可将它乘上 100，变成了“2341”一个整数。用键盘输入，再将输入的 ASCII 码转换为 2341 的内部整数表示格式。于是在机内，对于整数格式的各种操作，似乎全能用上了。当需要输出之前，再将机内结果除以 100 得到实际的结果。但是想象之中的许多问题也随之发生了。其中之一，就是精度问题。如果你的 23.41 元钱，加上得到的 7%的利息，你是拿到¥25.04呢?还是拿到¥25.05?如果不讲足够的精度的话，你拿¥25.04，恐怕要吃亏一分钱呢。

还有一种方法，叫定点数。但须在一个字节中，用 3 个位作为小数部分。例如：

$$10110110b = 2^4 + 2^2 + 2^1 + 2^{-1} + 2^{-2} = 22.75$$

这种方法，所能表示的数的范围不大而且是固定不变的(从 0 到 2^5)，增量为 2^{-3}。这些问题的解决，一定要靠浮点表示法。浮点格式可以使二进制小数点浮动。它有专门用来规定二进制小数点的位置的部分，还有用来表示数据有效数字的部分。在保证精度的同时，还可以有效地拓展数的表示范围。

浮点数实际上是靠两个整数来表示的，一个保持着数据的有效数字，一个规定了二进制小数点的位置。同时浮点操作可分解为作用于这些整数对的整数指令(加，乘和位移)序列。但是浮点运算光靠软件来完成，其速度慢得无法忍受，所以人们设计出许多硬件来直接模拟浮点运算。80387 就是 Intel 公司为用户奉献的支持浮点运算的数字协处理器。

1.2.2 IEEE 浮点标准

在深入研究 80387 的数据类型之前，我们想对 IEEE 浮点标准作一些说明。

早在 1979 年之前，就有许多大型的和微型计算机采用了浮点数据类型。然而当时却没有

一个浮点表示法的统一标准，以致于同一个 FORTRAN 程序，在不同的机器上运行，得出的结果也不一样。

在开发 8087 同时，Intel 公司对浮点的标准产生了兴趣，尤其是想开发一个供 8087 来实现的浮点标准。在 Intel 公司公布了它提出的标准之后不久，IEEE 成立了一个关于开发浮点运算标准的委员会。经这个委员会许多成员几年的辛勤劳动，产生了 IEEE 任务 P754 草案 10.0“二进制浮点运算标准”(Draft 10.0 of IEEE Task P754，“A Standard for BinaryFloating-Point Arithmetic”)。

草案中声明的该标准的部分宗旨如下：

(1) 促使已经存在于形形色色的计算机中的程序移植到遵循这个标准的机器上来。

(2) 增强编程者的才干和安全有效性，使那些虽不是谙熟数学方法的编程者也能产生高级的数学程序。

(3) 鼓励经验丰富者开发和奉献出实用而有效的计算程序，通过很少的编辑和再编译，移植到任何与该标准一致的机器上来。

这个 IEEE 标准被工业界广泛接受。它不但得到 Intel 芯片的支持，还得到许许多多微处理器、微型的和大型的计算机的支持。当然 80387 也不例外。

1.2.3 如果没有 80387 怎么办?

在一个 80386 系统中不含有 80387 是完全可能的，因为 80386 和 80387 是两个分离的硅集成芯片。若是你经常要处理大量的计算的话，可以用两种办法来解决：

(1) 在系统安装一块 80387。几乎所有的计算机制造商都会配备 80387 选件。

(2) 利用机内的数学模拟软件包。如果连软件包也没有，则设法配备或自编一个。

无论是用 80387 还是用模拟软件包，总是离不开为了 80387 设计的浮点数据类型和指令。也许你的系统上只有 80287，没有 80387。那么 80287 也可以运行，但要注意 80287 与 80387 的区别(请看附录 C)。

1.2.4 数据格式

表 1.4 列出了 80387 支持的七种数据类型。请注意它们表示的数的范围。80386 支持的最大整数为 2^{32}，或者 4.29×10^{9}(十进制)，这个数字很重要，要记住。

表 1.4 80387 支持的数据类型

数据类型	位	有效数字(十进制)	大致范围(十进制)
字的整数	16	4	$-32768\leqslant X\leqslant 32767$
短型整数	32	9	$-2\times10^{9}\leqslant X\leqslant+2\times10^{9}$
长型整数	64	18	$-9\times10^{18}\leqslant X\leqslant+9\times10^{18}$
压缩十进数(BCD)	80	18	$-99..99\leqslant X\leqslant+99..99$(18 位)
短型实数	32	6—7	$-3.39\times10^{-38}\leqslant X\leqslant3.39\times10^{38}$
长型实数	64	15—16	$-1.80\times10^{-308}\leqslant X\leqslant1.80\times10^{308}$
临时型实数	80	19	$-1.19\times10^{-4932}\leqslant X\leqslant1.19\times10^{4932}$

80387 也支持整数数据类型。于是一个既有实型数据又有整型数据的复杂计算只用一个 80387 就能完成，省掉了 80387 和 80386 之间的数据传送。

1.2.5 整型数

下面一张表列出了 80386 和 80387 支持的整型数：

位	80386	80387
8	带符号字节	不支持
16	带符号字	字整数
32	带符号双字	短整数
64	不支持	长整数

所有整型数都用二进制补码来表示。16 位和 32 位的整型数，虽同时得到两种芯片的支持，但不同的芯片对它们的称呼不同。图 1.8 表示了 80387 支持的三种整型数。字整型数可以表示从(－32768)到(32767)范围内的整数；短整数可以表示从(-2.147×10^{9})到(2.147×10^{9})范围内的整数；长整数可以表示从(-9.233×10^{18})到(9.233×10^{18})范围内的整数。

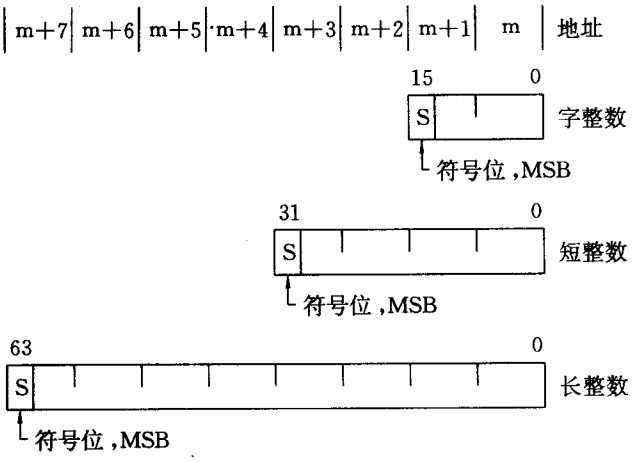


图 1.8 80387 的整型数(二进制补码)

1.2.6 BCD 码

80387 支持的是 80 位的压缩的二－十进制数据类型，可以保持 18 位的十进制数和一个符号位。如图 1.9 所示。BCD 码的 18 位数字，正好和 COBOL 的标准符合。虽然 80 位的 BCD 码中，有 7 位没有被利用，但也没有必要搞到 19 位，反而使得用起来复杂了。

1.2.7 实数格式

实数格式就是浮点格式。图 1.10 给出了实型数的一般格式。这种格式是由有效数字、指数、符号三部分组成。下面给出在三种实型数中，上述每一部分所占的位数：

数据类型	总位数	符号位	指数字段	有效数字段
短实型数	32	1	8	23
长实型数	64	1	11	52
临时实型数	80	1	15	64

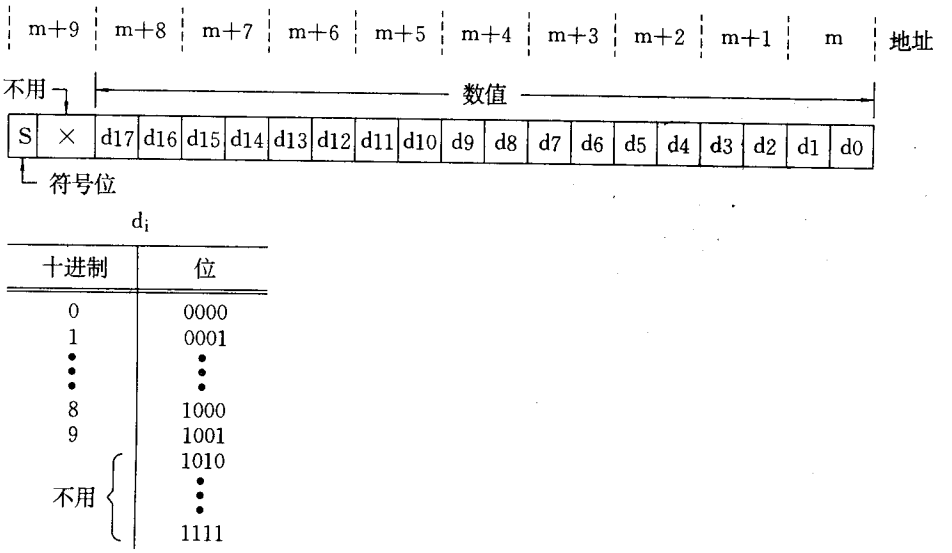


图 1.9　80387 的 BCD 数据类型

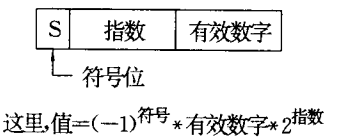


图 1.10　一般实型数格式

图 1.11 给出这三种数据类型的详细分布。若符号位为 1,表示该数为负,反之为正。

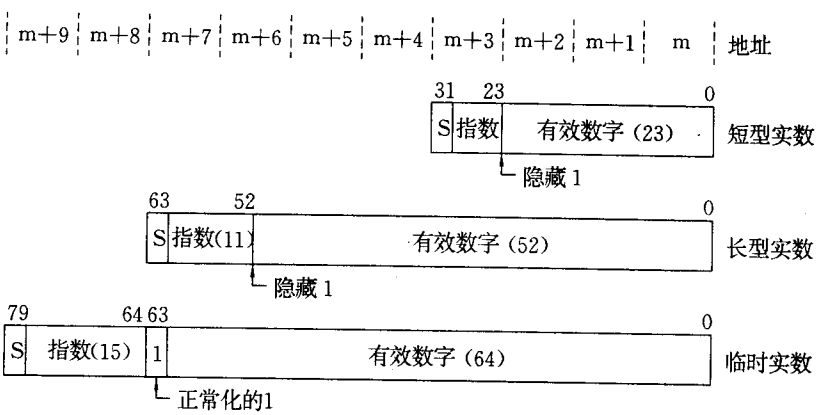


图 1.11　80387 的实数类型

有效数字有时被称之为尾数。下面我们就来说一说尾数的规则。在科学计数法中，同一个数可以有许多表示形式。例如：

$1 = 100.0 \times 10^{-2} = 10.0 \times 10^{-1} = 1.0 \times 10^{0} = 0.1 \times 10^{1} = 0.01 \times 10^{2} = \cdots$

在 80387 的实型数表示法中，虽然采用的是二进制，但上述问题是同样存在的。于是就需要做

一个规定。这个规定就是：任何实型数只能用下面的格式表示：

$$1.xxxxx \times 2^{n} \qquad (x \text{ 表示 1 或 0})$$

在二进制小数的左边，有且仅有一位，而且该位永远为 1。这样便产生了尾数的规则。

既然在二进制的有效数字中，小数点左边的一位永远是 1，那么我们只要承认这个事实，并将它永远“记在心中”就够了，不一定要把它“写”出来。省掉的这个“1”，还可以贡献出本应属于它的那个位空间，以提高表示近似数精度。80387 就在它支持的短实型数和长实型数的记数法中，压缩了小数点左边的这一位。在图 1.11 中看到的“隐藏 1”，就是这个被压缩了的位。于是，一个短的或长的实型数据，如果它的尾数为 0111…010b 的话，它的真正的有效数字应该是 1.0111…010b。指数字段是用偏移法来表示的带符号整数。在三种格式中，它们的偏移量分别为 127、1023 和 16383。例如，在短实型数中，指数 10000000b 对应为 2^{1}。偏移法和其它带符号整数表示法相比，有便于比较大小的优点。

图 1.12 可以和图 1.10 对照，它图解了我们刚刚解释过的浮点格式中的三个分量。

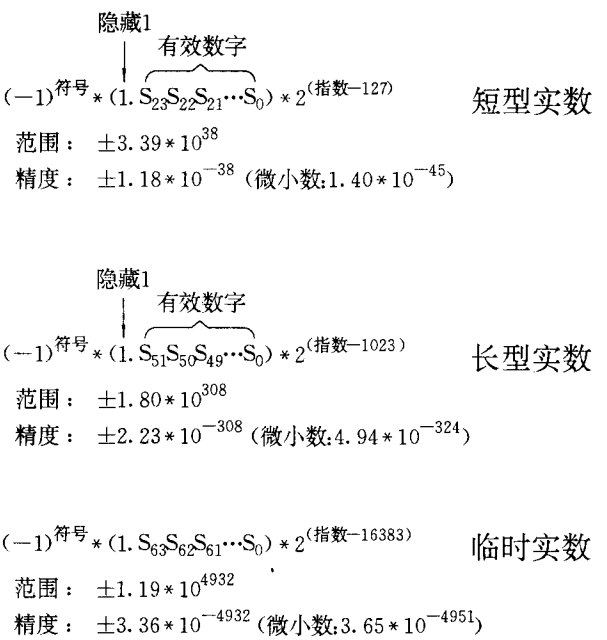


图 1.12　80387 支持的三种实型数的图解

图 1.12 中的“范围”给出了三种实型数所能表示的数值的范围(最大的绝对值)。图中的“精度”给出了三种实型数所能表示的最小正负数(最小指绝对值)。“微小数(Denormals)”留到以后再作解释。下面的例子都以短实型数的格式给出：

符号位	指数位	有效位	含义	值
0	10001110b	000100…0b	$(1+2^{-4})\times 2^{15}$	34816
1	01111100b	011000…0b	$-(1+2^{-2}+2^{-3})2^{-3}$	−0.17188
0	01111111b	000000…0b	$(1)\times 2^{0}$	1

1.2.8　临时实型数

我们上面讨论过的三种实型数格式，除了表示法的三个部分的位数不同以外，其余都十分

相似。但临时实型数格式和它们就不同了：

(1) 临时实型数格式是 80387 内部的格式。无论你给出什么样类型的数(短整数、BCD、短实型数等等)，在 80387 内部都将它们转换成临时实型数。

(2) 临时实型格式没有隐藏位。一个格式化的数的临时实型数格式的第 63 位永远为 1。80387 采用此临时实型数，最大限度地提高了运算精度，扩大了运算范围。假如你用短实型数做一个计算，往往会碰到中间结果大大地超出短实型数的表示范围的情况，尽管最后的结果仍然用短实型数来表示。

因为临时实型数在 80387 内部要直接参加运算。所以如果有"隐藏 1"的话该请它出来参加运算了。临时实型数的第 63 位，就是这位隐者出场的地方。当短型或长型实数格式在 80387 内部转换成临时实型格式时，第 63 位就自动插入"1"。

1.2.9 特别的数

80387 支持的三种格式(短实型数、长实型数和临时实型数)，在机内可能出现的数据表示，用表 1.5 到表 1.7 列出。

表 1.5 短的和长的实型数表示

符号	偏移指数	有效数字	种类
0/1	11…11	11…11	静的 NaN
0/1	11…11	⋮	
0/1	11…11	10…00	
0/1	11…11	01…11	信号的 NaN
0/1	11…11	⋮	
0/1	11…11	00…01	
0/1	11…11	00…00	无穷数
0/1	11…10	11…11	正常数
0/1	⋮	⋮	
0/1	00…01	00…00	
0/1	00…00	11…11	微小数
0/1	00…00	⋮	
0/1	00…00	00…01	
0/1	00…00	00…00	零

表 1.6 临时实型数表示

符号	偏移指数	有效数字	种类
0/1	11…11	111…11	静的 NaN
0/1	11…11	⋮	
0/1	11…11	110…00	
0/1	11…11	101…11	信号的 NaN
0/1	11…11	⋮	
0/1	11…11	100…01	
0/1	11…11	100…00	无穷数
0/1	11…10	111…11	正常数
0/1	⋮	⋮	

续表

符号	偏移指数	有效数字	种类
0/1	00…01	100…00	
0/1	00…00	111…11	伪微小数
0/1	00…01	⋮	
0/1	00…01	100…00	
0/1	00…01	011…11	微小数
0/1	00…01	⋮	
0/1	00…01	000…01	
0/1	00…01	000…00	零

表 1.7　不支持的临时实型数表示

符号	偏移指数	有效数字	种类
0/1	11…11	011…11	伪 NaN
0/1	11…11	⋮	
0/1	11…11	000…01	
0/1	11…11	000…00	伪无穷数
0/1	11…10	011…11	非正常数
0/1	⋮	⋮	
0/1	00…01	000…01	
0/1	11…10	000…00	伪零
0/1	⋮	000…00	
0/1	00…01	000…00	

在这些表中，除了在各种表示法的正常数据范围内的正常数(Normals)以外，我们还可以找到零(Zero)、微小数(Denormals)、无穷数(Infinities)、信号的非数字(SignalingNaN)、静的非数字(Quiet NaN)等，这些特别的数。下面我们就来逐一讨论这些数。

1. 零

零的偏移指数为 00…00b，有效数字段亦为 00…00b。零的偏移指数是保留的，也就是说，零的偏移数不能用来表示正常的实型数。并且注意，只在零作除数时，零才有正负之分，否则正零和负零没有区别。这一点，在第 3 章介绍 FDIV 指令时还会谈到。

2. 无穷数

无穷数的偏移指数 11…11b 也是保留的，不用来表示正常的实型数。无穷数的有效数字段全是为 0。无穷数分正无穷和负无穷。

3. 微小数

微小数用来表示非常小的数。在许多计算机中，从最小的可以规格化的数到零的表示法之间，没有任何形式的过渡。即，比最小的规格化的数再小一点的数，便是零的表示式。我们称这种从最小数到零的下溢为“陡然下溢”。80387 则不然，它为最小的正常数和零之间提供了平滑或渐变，它在两者之间插入了一系列从小到大的表示比零大，比最小正常数小的“微小数”表示法。80387 的这种下溢，我们称之为“逐渐下溢”。逐渐下溢使用的表示法，叫做微小数表示法，或非规格化的表示法。这种表示法会引起精度丢失，但它有效地拓展了能表示的非常小数的范围。很显然，尽管逐渐下溢要丢失一些精度，但还是比陡然下溢要优越得多。

在正常情况下，数字需要规格化(左移，直至有效数字的最高有效位为1)。然而，微小数的有效数字的最高位为0，它不符合规格化的要求。微小数的偏移指数为00…00b也是2^{-126}，2^{-1022}，和2^{-16382}的特殊表示形式。说它"特殊"，是因为2^{-126}，2^{-1022}和2^{-16382}的正常偏移指数为00…01b。微小数总会引起精度丢失；否则的话，便使用正常数。从图1.12上看出，利用微小数可以有效地表示非常小的小数。

4. 伪微小数(Pseudo Denormals)

80387支持伪微小数，但不会产生伪微小数。伪微小数和微小数的区别在于前者的有效数字的最高位为1，后者的有效数字的最高位为0。伪微小数的有效数字是符合规格化的要求的。伪微小数的偏移指数仍为00…00b。我们曾经说过，偏移指数00…00b和00…01b是等值的，都对应着2^{-126}，2^{-1022}，和2^{-16382}。所以说伪微小数可以用，但没有用规格化的格式表示。这就是它的特别之点。伪微小数是用微小数格式表示的正常数。

5. 非数字(NaN——Not a Number)

NaN——仅仅具有数字形式，但它并不是数字。NaN分为两大类：信号的SNaN和静的QNaN。当在操作中使用了SNaN，会引起非法操作异常；而在操作中使用了QNaN时，不会引起异常。这就是它们的区别，也是它们的命名的原因。

SNaN在短实数和长实数中，最高有效位为0，在临时实数中最高有效位是隐藏位为1，故第二位为0。

80387并不产生SNaN，而是编程器或编译器先将程序中的所有变量初始化为SNaN。于是若有变量在赋值前就被利用的话(此时该变量值仍是一个SNaN)，非法异常便产生了。用这种方法，就能检查出那些未赋值而使用的变量，这是应用SNaN的一个例子。NaN能在其有效数字段的小数部分存放关于产生这个NaN的场所和原因的各种信息。QNaN在短实数和长实数中，最高有效位为1，在临时实数中，最高有效位为1，且第二位也为1。QNaN产生于非法操作异常发生时；当这种异常发生，产生的结果是不定的(在下面给出)。在所有情况下(除了第3章要讨论的FCOM、FIST和FBSTP)，QNaN和SNaN被保留在操作的全过程中。在处理NaN时有下面几种特殊情况：

(1) 引用了NaN的无效操作异常，将产生不定的QNaN。整数、BCD和实数的不定在下面给出。

(2) 当操作含有SNaN和QNaN，其结果为QNaN。

(3) 两个SNaN之间的操作将产生一个更大的SNaN，然后再转换为QNaN。这种转变很简单，只要把有效字段的最高位置1就行了。

(4) 两个QNaN之间的操作将产生一个更大的QNaN。

(5) 一个SNaN和一个正常数之间的操作将产生一个SNaN，再转换成QNaN。

(6) 一个QNaN和一个正常数之间的操作将产生一个QNaN。

为了更清楚地说明上述结果，表1.8总结了有关NaN的操作。表中假定对操作数1和操作数2(op_1和op_2)发生了操作。

不确定值是上述静态NaN的特例。80387支持的每个数据类型(字，短的和长的整数，BCD和短长临时实型数)有其唯一的不确定值的表示法。当一个非法操作异常发生并且输入操作数中没有NaN时，80387就产生这种表示法。产生非法操作的指令在第3章中给出。非法异常如何产生在第5章中讨论。表1.9总结了每种数据类型的不确定值的编码。编码中最左边的一位是最高有效位。对于实数的表示法，从左到右分别是符号字段、指数字段以及有效

数字字段。

要注意，当我们讨论 BCD 数和整数类型时，我们没有说出全部情节。因为我们尚未提到不确定值。对整数数据类型，当产生不确定值时，就产生最负表示法。而对于 BCD 数则用未用过的编码来表示。

对于实数格式，可以装载和存储不确定值。因为实数的不确定值属于 NaN，当装载一个不确定值时是不会产生异常的。对于整数和 BCD 数，是不能装载不确定值的。对于整数，当装载不确定值时是被当作最大负数来处理的。对 BCD 数装载不确定值的结果没有定义。

表 1.7 给出的不被支持的临时值数据的表示法纯粹是因为 80387 不支持之故(有些是被 80287 所支持)。如果 80387 遇到这些数据格式将引起非法操作异常。

表 1.8　含有 NaN 的操作

		操作数 2		
		Q	S	R
	Q	>Q	Q	Q
操作数 1	S	Q	CQ(>S)	CQ(S)
	R	Q	CQ(S)	QI

关键字	
Q	静的 NaN
S	信号的 NaN
R	规则实数(非 NaN，合法数)
>Q	操作数 1 和操作数 2 的较大的 NaN
>S	操作数 1 和操作数 2 的较大的信号 NaN
CQ(X)	把 X 转换到 QNaN
QI	默认的静的 NaN—不定的实数

表 1.9　不确定值的编码

形　式	位	编　码
字整数	16	100…000b
短型整数	32	100…000b
长型整数	64	100…000b
BCD	80	1 1111111 1111 1111 XX…XXb
短型实数	32	1 11…11 10…00b
长型实数	64	1 11…11 10…00b
临时实数	80	1 11…11 100…00b

注意：X 表示不关心

1.2.10　异常

异常表示在当前操作期间已经检测到了某种类型的错误。例如，当你发动汽车时，如果车门还敞开或座位保险带没有系紧，就会响笛。两种情况都表示检测到了异常。

有许多可能的异常。例如上面所说的精度的丢失，取决于机器的状态，精度的丢失可能被当作异常来处理。我们也提到过信号的 NaN。对一信号的 NaN 的操作将产生别的异常。有许多其它可能的机器的异常。在第 5 章中将讨论异常条件的详细情况以及如何进行这些处理。

第 2 章　机器状态和存储器寻址

第 1 章叙述了 80386 和 80387 能识别的数据类型，以及在存储器中如何存储这些数据的。80386 和 80387 的机器指令直接对这些基本的数据类型进行操作。每条机器指令指定了要执行的操作，以及参与操作的输入或输出的数据的位置。输入或输出的数据叫做操作数。

本章叙述如何寻址由 80386 当作机器指令操作数的基本数据类型。操作数可以在 80386 处理器的寄存器中，或者在处理器外的主存储器中，或者在 I/O 存储器中，或者在指令中作为程序立即常数的数据。主存储器提供了一个非常大的存储器操作数的空间，但是访问主存储器比起对寄存器或立即常数的访问要慢得多。更进一步，有些操作要求某些操作数是在寄存器中，而不是在主存储器中。

本章首先叙述应用程序员可用的三种类型的寄存器：用于存储 32 位数的寄存器，用于控制处理器的执行的寄存器，用于寻址内存的段寄存器。接着我们叙述用来构成存储器地址的方法以及在指令组中操作数的二进制编码。然后介绍独立的 I/O 存储空间。最后叙述在 80387 浮点处理器中可以使用的存储浮点类型数据的寄存器，以及控制浮点指令操作的寄存器。

2.1　寄　存　器

应用程序员可用的寄存器组由 16 个寄存器组成，如图 2.1 所示。这些寄存器可以分成三类。以后再叙述的寄存器是系统程序员使用的，第 4 章叙述了支持存储器管理的寄存器，附录 A 叙述了支持程序调试的寄存器。

这三类寄存器叙述为：

(1) 用于算术和逻辑运算的 8 个 32 位的通用寄存器，它们也可用于基址和变址寻址时存放地址。

(2) 两个处理器控制寄存器。

(3) 6 个 16 位的段寄存器，它们用来寻址存储器的段。每个寄存器一次只提供直接访问一个段的内存。内存的段的含义在本章的后面加以说明，而在第 4 章中将加以详细说明。

80386 的寄存器组是先前的 86 系列处理器所用寄存器组的超集。先前的 16 位寄存器被扩展成 80386 上的 32 位寄存器。386 寄存器的名字是由老的 16 位寄存器的名字之前冠以 E 组成。例如，8086 中 16 位的 AX 寄存器扩展成 80386 中的 EAX 寄存器。16 位的 IP 寄存器扩展成 386 中的 EIP 寄存器等等。

2.1.1　通用寄存器

80386 有 8 个 32 位的通用寄存器，用于诸如加减乘除等的算术运算以及形成内存的地址。这在本章后面将会叙述。这 8 个寄存器定名为 EAX、ECX、EDX、EBX、ESP、EBP、ESI 和 EDI。见图 2.1 所示。

这些寄存器的低 16 位可以作为 16 位的寄存器独立访问。并且把它们命名为 AX、CX、DX、BX、SP、BP、SI 和 DI，这些就是 86 系列处理器的以前的成员的 8 个 16 位通用寄存器。这