北京希望
电脑公司

80×86和80×87技术参考手册

晓军 冀伟 编著

海洋出版社

80×86和80×87技术参考手册 晓军 冀伟 编著

北京希望电脑公司高级程序设计丛书

80X86和80X87技术参考手册

晓 军 冀 伟 编著

海洋出版社

1992·北京

内 容 简 介

随着软件日益完善，功能日益增强，系统软件程序员不可避免地要涉及到计算机的心脏--80X86处理器和80X87协处理器结构．只有全面了解80X86的体系结构，系统软件程序员才有可能更进一步地利用80X86处理器的特性编制更强，更完善的软件．

本书是一本关于80X86处理器和80X87协处理器结构和指令的参考手册．它既可以做为一个指令集供临时的汇编语言程序员查找，也可以为那些想要进一步了解80X86和80X87结构细节的系统软件程序员提供全面、系统的描述．

在本书中采用了C语言符号描述80X86和80X87的全部特征，由于C语言符号为系统软件程序员所熟悉且它相对精确，所以这有助于系统软件程序员更快和更深地理解80X86和80X87结构和指令集．

本书涉及到了80X86和80X87的所有主要成员，包括8086、8088、80186、80286、80386、80387和i486，以及与其对应的各80X87成员．在第二章中详细地描述了80X86的发展过程及其各过程的主要成员，各主要成员在性能上的改善和增强以及所具有的指令集，着重指出了各成员具有的独特功能和特性．第三章与第二章类拟，它详细描述了80X86配套产品80X87的发展过程和各成员的主要特性．若想了解80X86和80X87的指令，则参考第四章，它按字母顺序全面、系统地列出了所有80X86和80X87成员拥有的指令，并且给出了其定义、解释、C语言描述和例子，而非指令的结构概念，象中断和异常处理及80X86和80X87的接口在第二章和第三章中给出．

需要本书的用户可直接与北京8721信箱联系，邮政编码100080，电话2562329．

(京) 新登字第087号

80X86和80X87技术参考手册

晓军　冀伟　编著

责任编辑　　阎世尊

*

海洋出版社出版

北京市新华书店发行　施园印刷厂印刷

*

开本：787×1092 1/16　印张：26.25　字数：584 千字

1992年9月第1版　　1992年9月第1次印刷

印数 1-3000

ISBN 7-5027-2877-5/TP·125

定价 15.00 元

目　录

第一章　引言 …… 1
1.1　设备更新 …… 1
1.2　如何使用本书 …… 3
1.3　符号 …… 3

第二章　80x86 机器 …… 7
2.1　80x86 机器 …… 7
2.2　章节组织 …… 9
2.3　数据表示 …… 9
2.4　寄存器组 …… 12
2.5　执行方式 …… 24
2.6　保护和内存管理概念 …… 25
2.7　存储器访问规则的优先级 …… 34
2.8　堆栈操作 …… 78
2.9　I／O 访问 …… 82
2.10　门和优先级转换 …… 84
2.11　任务切换 …… 89
2.12　中断和异常 …… 101
2.13　指令可中断性 …… 116
2.14　指令再启动性 …… 117
2.15　停止和运转停止 …… 118
2.16　软件调试特性 …… 118
2.17　处理器初始化 …… 124
2.18　指令译码 …… 131
2.19　80x86 成员的差异 …… 141

第三章　80x87 处理器扩展 …… 146
3.1　IEEE 浮点标准和 80x87 …… 146
3.2　本章的结构 …… 147
3.3　80x87 数据表示 …… 147
3.4　IEEE 浮点运算原理 …… 156
3.5　80x87 综述 …… 160
3.6　80x87 寄存器 …… 160

3.7 80x87 堆栈操作 …… 171
3.8 80x86 和 80x87 的协调 …… 172
3.9 80x87 异常报告 …… 174
3.10 模拟数字处理器扩展 …… 178
3.11 80x87 初始状态 …… 178
3.12 80x87 指令编码 …… 179
3.13 80x87 检测方法 …… 180

第四章 指令集 …… 181
4.1 按功能排列的指令助记符 …… 181
4.2 符号 …… 186
4.3 按助记符分类的指令 …… 191

附录 A 按操作码分类指令 …… 397

试读结束：需要全本请在线购买： www.ertongbook.com

第一章 引言

本书是一本关于 80x86 的参考手册. 80x86 处理器的结构体系用于 PC 和 PS／2 系列微型计算机中. 80x86 拥有多种实现形式, 但并不是所有都支持这里描述的体系结构. 确实, 在这十年中, 随着半导体设计技术的进步, 体系结构也不断发展. 80x86 系列的最新成员, i486, 就是提供广泛的内存管理和保护功能的 32 位处理器. 同时, 它还提供内部浮点计算支持以及超高速缓存功能. 与最始的 8086 成员 8086 相比, 8086 仅支持 16 位软件体系结构且没有内存管理或保护特性. 8086 上的浮点运算必须经一个独立的协处理器 (8087) 完成. 8086 无集成化超高速缓存; 既然普通存储器速度快于其执行速度, 所以不需要超高速缓存. i486 比 8086 大约快 30 到 40 倍. 8086 和 i486 与其它 80x86 体系结构中的成员一样都是 Intel 公司的产品 .

如上面讨论所示, 80x86 的演化过程是一个向上兼容的扩展过程, 它源于 8086, 8086 后面的处理器标志已发展到 80286, 80386 和 i486. 这些处理器各自主要特征在图 1-1 中给出. 直到 80386, 80x86 系列中每个成功的一代产品都在其上代产品上增加了显著的结构特征. 特别是, 80286 在基本的 8086 结构上增加了面向段的内存管理和保护功能, 80386 扩展结构到 32 位且增加了页内存管理和保护.

产品功能伴随着每代产品而逐渐增强的趋势到 80386 而截止. 80386 的继承者, i486 已把重心移至性能改善和集成化,而不是结构上的增强. 在 i486 中, 唯一新指令是 BSWAP, 该指令颠倒双字节 (32 位) 中字节的次序. 可以预测, i486 的继承者将会遵循这个发展趋势.

1.1 设备更新

尽管在图 1-1 中仅列出了 80x86 的四代产品, 但 Intel 公司目前销售 9 种不同的类型的 80x86 成员. 然而, 这些附加的处理器只是列出的主要设备的变型. 每种变型在下面的段落中将会简要介绍.

8086, 拥有一个 16 位存储器接口, 它有一个变更类型, 即 8088, 该类型有一个 8 位存储储器接口. 减少存储器接口宽度减少微机中需要的器件数, 而且降低了复杂程度和系统造价. 在最初的 IBM PC 机中使用了 8088 处理器.

80286 和 i486 没有变型. 可是, 80386 拥有两种变型: 80386SX 和 80376. 80386SX 可以看做是 80386 的 8088, 在 80386 和存储器间正常的接口宽度为 32 位. 在 80386SX 中, 这个宽度降到 16 位. 在软件功能和数据路径设计术语中, 80386SX 等同于 80386. 80376 是一种怪型. 它类似于 80386SX, 因为它拥有一条 16 位总线; 可是, 在 MS-DOS 和通常的编程中不允许使用它. 80326 主要用于 32 位控制器应用中. 通过保留 80386 结构的附属设备, 80376 程序能够使用用于 80386 的编程工具库. 有关 80376 的详细内容请

看第二章。

处理器	推出日期	主要性能
8086	1978	16 位软件结构 16 位直接（偏移）寻址 用于浮点的 8087 协处理器 近 0.33MIPS
80286	1982	8086 功能+ 扩展的段基址保护结构 用于浮点数的 80287 近 1MIPS
80386	1985	80286 功能+ 32 位软件结构 32 位直接（偏移）寻址 页内存管理和保护 用于浮点数的 80387 近 3.5MIPS
i486	1989	80386 功能+ 集成化的 80387 式协处理器 集成化的 8KB 指令／数据超高速缓冲存储器 多处理器支持 近 12MIPS

图 1-1　主要 80x86 产品的主要性能

我们已经论述了 80x86 系列成员中的三种变型．另外一个系列成员——8086，并不真是一种变型，而是一种独立的设计，它用来改善 8086 以及集成象中断的控制器和 DMA 那样的外部设备．因为这些外部设备的配置不能与 IBM PC 的系统结构相配套，所以 80186 不能被普遍应用于 PC 兼容的设计中．像 80376 一样，它的主要市场是控制器应用．80186（它有 16 位宽的内存接口）也有种变型-80188，它拥有一个 8 位宽的内存接口。

此外，对于不同设备间的差异，还有一点也值得注意；大部分设备有不同的频率．例如，8086 的最大工作频率范围从 5MHz 的 10MHz，80286 的最大工作频率范围从 6MHz 到 20MHz，而 80386 的最大工作频率范围从 16MHz 到 33MHz．目前，i486 能提供一种

25MHz 的部件，此后会有 33MHz 的部件。在一种特定的设备中，一种部件的频率直接关系到 CPU 的性能。然而，设备间的结构差异也会影响 CPU 的性能；例如，25MHzi486 的性能是 25MHz 80386 的两倍。此外，系统级的性能（即可被计算机用户觉察的性能）要受附属于 CPU 的存储器系统的速度和它的 I／O 装置的性能所影响。

1.2 如何使用本书

尽管本书基本上是一本参考手册，但是它的编写是为了两种人的使用。一个是查找特定指令定义的临时的汇编语言程序员；另一个是想要进一步了解有关结构某些细节的系统软件程序员。这些目标的达到在很大程度上依赖于改进的 C 语言符号去描述这个结构的全部特征。想要给简单指令定义的用户可以仅参考在第四章中按字母顺序列出的指令。（第一次使用的读者最好读一下这章的介绍部分。）在第四章中的定义通常比较简洁。如果一个特定指令完成复杂的控制序列，它的描述在子程序调用中总是被压缩。所有这些子程序的定义会在第二章中给出。这章还将描述非指令的结构概念，像中断和异常处理。

第四章提供了关于 80x86 和 8087 指令的定义。然而，80x87 许多特征的最好描述并不包括在这些指令定义的内容中。如，由 80x87 指定的 80x86–80x87 的接口和控制浮点运算器的一般原理。这些细节将在第三章中给出。

正文描述的线性特征和 80x86 的结构并不一致。它包括一个互相关概念网络。本书将通过提供有关描述的广泛的交叉参考信息来解释这个问题。读者最好先查阅索引以便确知本书各部分所描述的主题。

1.3 符号

为了更加精确和简洁，常常使用 C 编程语言来描述结构。即使 C 语言符号有时是模糊的，但我们反对任何方式的临时改变。可是，标准的 C 语言符号已加强功能；例如，允许位字段选择。不熟悉 C 语言的读者应该考虑选择一种较好的语言文本；例如，看 [Kernighan and kitchie, 1978]。下面一节描述了在本书中使用的 C 语言扩展符号。

1.3.1 C 的符号扩展

C 语言扩展为在一个基本 C 类型中允许位字段处理。例如：

```
int i;
char b;

b = i(9:7);
b = signex(i(9:7));
i(20:15) = b(6:1);
```

第一个赋值是把 i 中从下标 7 到下标 9 的 3 位字段的值拷贝到字符 b 中。b 中的高位（如，3 位到 7 位）清零。第 2 行完成同样的赋值 （区别为它假定两字段为互补值）。第三行拷贝 b 的 1 至 6 位到 i 的 15 至 10 位，i 的剩余位保持不变。注意，这里的所有值都

是按 Little Endian 符号编码的，因而值的最小有效位总是带有最小的位下标。

另一个差异是如何使用算术和关系算子。在标准 C 语言中，实际算子工作依据操作数的类型。例如，程序段：

```
int i1, i2, i3;
unsigned int u1, u2;

if (i1 > i2)
   i1 = i2 / i3;
if (u1 <= u2)
   u2 = u1-1;
```

按符号值比较 i1 和 i2（既然 int 是整数），且 U1 和 U2 按无符号值比较。因而，i2 除 i3 的商是按假定的符号除生成。对于灵活性而言，我们需要准确的操作类型规范。特别是，无符号运算符带有一个下标“n”，这里有符号操作符未采用。因此，上面的无符号比较修改为

```
if (u1 <=u u2)
   u2 = u1-1;
```

这里有符号除不变。

在本书中无符号运算符的完整清单如下所示：

- $/_u$：无符号除（生成正商）
- $\%_u$：无符号取模（生成正模数）
- $>_u$：无符号“大于”
- $<_u$：无符号“小于”
- $<=_u$：无符号“小于或等于”
- $>=_u$：无符号“大于或等于”
- $*_u$：无符号乘
- $>>_u$：逻辑右移，空出位填零。（算术右移运算符“>>”使用原移位操作数的符号填充空出位）。

所有表达式按假定的无限精度计算。这样做是为了符号的便利。以 80x86 设备来说，无限大表达式计算很容易通过溢出条件检测，而有限表达式计算会发生溢出。而且，由两个单精度值很容易生成双精度值。一个表达式的值总是被截断以适合目标所要求的位数。有时，它对于一个无限精确目标也很有用。特定变量 result 就是用于这个目的。例如：

$$result = dst - src$$
$$OF = result < -2^{os-1} \ || \ result > 2^{os-1}-1$$

变量 dst 和 src 的无限精确差被赋值给 result。当标记被认为是带符号的量时，如果 result 不适合目标，它将置 OF 标志为 TRUE。如果 result 不表示无限精度的值，定义 OF 值的表达式将会比较复杂。当然，如果 result 被赋给有效长度的对象，则仅是它的最少有效位被拷贝。例如，

EAX = result

拷贝 result 的最少有效 32 位到 EAX，假定 EAX 是 DWORD 的存储单元。

有时，上面提到的扩展 C 符号是不足以描述某种概念的。当这类情况发生时，附加 C 概念要在文中被描述，并用 <<……>> 括起来。例如，

```
if ( « NMI pin active » )
   handle_intr_xcp(2, FALSE);
```

激活 NMI 脚导致被 handle__intr__xcp 指定的一系列活动。

最后，注意的是在临时变量的说明上 C 语言符号有一点小的自由。有时，这些变量（通常存储中间结果）不被说明。这种情况是很少的，因为这类临时变量是依据上下文产生的。

1.3.2 未定义和保留字段

有时，结构不说明如何修改某种状态。例如，结构指明在乘法算子后零标记（ZF）的值是未知的。这是通过赋给它未定义的特定值来表示的。例如，

ZF = undefined

尽管在 80x86 结构的某个成员中"未定义"的值可能产生同样的位模式，但程序不应把一个值与未定义值作比较，例如，在某个 80x86 成员中，在乘法算子后我们把 ZF 总置位于零。现在，如果把 EFLAGS 寄存器（包括 ZF）的值与某些常数作比较（如，检测 OF 标记），在不同 80x86 结构的成员中就会产生不可预测的结果。为了比较 EFLAGS，唯一可靠的方法是在比较前把未定义值还原。

除了未定义表达式外，结构有时把下面各项标为保留状态。

- 一个完整的寄存器（如，控制寄存器 CF4）
- 寄存器中一个或多个字段（如，EFLAGS 寄存器的高位 13 位）
- 字段的特定值（如，控制描述符 Type 字段的符号 8）

因此，保留项是为了未来的可扩展性而做的标记。他们还用于特定的 80x86 成员的某些特性，而对于通常用户来说明是毫无用处的，例如，执行内部可测试性特性。因此，保留寄存器并不被读出和写入。当所读寄存器是一个保留字段时，保留字段的值被认为是未定义的。当修改寄存器中的某一字段时，而它也包括保留字段，按以下步骤进行：

1. 读寄存器到临时寄存器。（在大多数情况下，临时寄存器是一个通用寄存器，像 EAX）。

2. 修改临时寄存器的位数，与被修改的原始寄存器的字段相对应。

3. 把临时寄存器作为一个整体写入原始寄存器。

这些步聚保证了所有保留字符的值未被修改。

1.3.3 过程描述的问题

在本书中用伪 C 语言去描述 80x86 的结构，这是因为它的符号比较熟悉和相对精确。然而，在任何过程符号（像 C）中都会产生一个严重的缺陷，即描述产生一系列活动。这种强制的次序在说明中是不需要的。本书采用的方法是使用 C 语言，而不是使用更替符号，但是还需作下列说明。

- 如果I_1和I_2是80x86依次执行的指令，设备必须保证在I_2开始前I_1结束。这个指令的全部要求只是考虑到软件的运行。特别地是，这项规则不能清除如流水线技术那样的重叠指令处理技术的使用。
- 如果80x86指令产生多个存储器访问，那么以C语言表达的指令行为规定存储器访问次序，与 80x86 特定设备将执行访问的次序并不样。然而，应该注意的是产生的存储器访问次序总是被保存在指令级。特别地，假定$_1$，产生存储器访问 m_{11} 和 m_{12}，I_2 产生存储器访问 m_{21} 和 m_{22} 和 m_{23}。假定他们没有交互，存储器访问 m_{11} 和 m_{12} 可以按任何顺序执行，m_{21}，m_{22} 和 m_{23} 也一样。然而，存储器访问 $m_{21}-m_{22}-m_{23}$ 必须在存储器访问 $m_{11}-m_{12}$ 后执行。
- 80x86具有按照存储器访问在一个指令中所执行的顺序进行选择的功能，这种功能会导致不同的异常报告。例如，如果存储器单元 a 和 b 在指令中被访问，并且两个单元的访问引起 PAGE 错误，由特定成员报告的页错误地址会是 a 和 b 中的一个。

试读结束：需要全本请在线购买： www.ertongbook.com

第二章　80x86 机器

我们的目标是像一个程序一样具体分析 80x86 结构。为了达到这个目的，我们必须具体分析 80x86 数据结构（也就是，80x86 寄存器和存储器）以及在这些数据结构上工作的算法。算法组成一定数量的指令和处理中断和异常情况的方法。对于指令的描述而言（见第四章），在本章中将描述 80x86 结构的各个方面。但浮点支持描述，由于除了 i486 以外，在所有 80x86 设备中它都是使用一个独立的协处理器，所以这一部分将放在第三章中描述。

2.1　80x86 机器

首先，先检查 80x86 结构的执行级循环。

```
x86_cpu()
{   reset_cpu();                        /* see page 178 */

    while (1) {  /* do forever */
       checkpoint();                    /* see page 168 */
       instr = decode_instruction();    /* see page 185 */
       execute(instr);                  /* see page 259 */

    NEXT_INSTR:
       report_brkpts();                 /* see page 175 */
       check_interrupts();              /* see page 151 */
    }
}
```

80x86 以 reset__cpu 例行程序开始工作，这个例行初始化大多数 80x86 寄存器以便于进入一个程序执行可以开始的状态。reset__cpu 例行程序在 2.17 中描述。在 80x86cpu 初始化后，输入指令中断循环。这是一个无限循环，它仅在 cpu 复位时中止。

80x86 指令译码在下面五部分中完成。首先，检查程序员可见的寄存器。由检查点 (checkpoint) 例行程序完成的检查存在辅助寄存器中。既然它在当前指令执行前完成，则它仅反映上条指令执行后 CPU 寄存器的状态。这个检查允许执行当前指令故障时恢复。故障恢复和检查点例行程序在 2.14 中详细讨论。注意应用程序不必担心故障恢复。操作系统软件结合 80x86 采取的动作使故障恢复成为透明的。

在检查点例行程序执行后开始处理当前指令。指令以紧凑的、编码的表示存在存储器中。当前指令从存储器中取出且由 decode__instruction 例行程序解码（在 2.18 中描述）。这个例行程序也使当前指令指针（称为 EIP）指向下条指令。decode__instruction 例行程序决定要完成的操作类型和其参数。参数包括指令操作数（也就是，寄存器名和／或存储器地址），如果需要，也包括操作数的长度（8，16 和 32 位）及是否需要自动完成存储器

访问。操作类型和参数供给 execute 例行程序，它自动完成操作数规定的操作。如果在完成操作数检测到异常情况，则执行相应的异常情况处理程序。既然指令的描述是本书的主要内容，所以，它单独组成一章内容——第四章。

80x86 结构支持软件调试程序中使用的多种特性。它们是：

1. 指令单步执行
2. 指令地址断点
3. 数据地址断点

单步执行允许用户指定的程序单条指令地执行。指令断点和数据断点允许用户程序执行一组指令或访问一组数据地址。这些状态的检查由 report__brkpts 例行程序完成。这个例行程序在 2.16.3.4 中描述。注意所有的 80x86 结构的成员仅都支持单步执行。指令和数据断点仅对 80376，80386 和 i486 有效。

指令译码采取的最后行动是检查外部中断（也称硬中断）。这个功能由（check__interrupts 例行程序完成，它在 2.12.5 中描述。

在中断检查以后，对于下条指令重复执行完整的五步工作。注意，了解"下条"指令不必是跟在当前指令后面的指令很重要。例如，当前指令指针在遇到分支时将由 execute 例行程序改变。report__brkpts 例行程序将在检测到断点时更新指令指针为调试服务例行程序的首条指令处。check__interrupts 例行程序将在发生中断时使指令指针指向对应的中断处理程序入口处。在后两种情况下，在伴随 execute 例行程序完成的指令指针地址存程序堆栈中以便在调试服务程序或中断处理程序完成以后，断点和中断前的执行线索可以恢复。

2.1.1 指令覆盖

80x86 结构要求其所有设备在下条指令开始前当前给定的指令应该完成。可是，这个要求在完成存储器和 I／O 访问以及断点报告时将放宽。特别是，80x86 设备允许在来自先前指令的存储器（或 I／O）访问完成以前开始一条新的指令执行。（在下条指令开始前所有存储器故障检测仍然需要完成。这保证了一个故障报告的指令地址总是与存储器访问遇到的指令地址匹配。）

允许存储器或 I／O 访问物理完成延迟的结论是仅 CPU 内部动因才影响程序的行为。特别是，假定一个基于 80x86 系列处理器的系统在承认某个 I／O 寻址时观察 I／O 寻址和产生一个硬中断（这可以完成，例如：在 80286 上模拟 80386 和 80486 的 I／O 允许位图。）在这样一个系统中，中断处理程序不必带指向发出 I／O 命令指令的下条指令的寻址输入。这是因为 I／O 寻址对外部系统而言是可见的，80x86 可能已开始执行下条指令。

当使能数据断点检测时，情景与上段描述的类似。数据断点通常按陷阱报告；也就是，在执行产生数据断点指令的下条指令以前报告。可是，如果由于在一条指令结尾处产生存储器访问而检测断点，则下条指令可以在报告断点前完成。提供这个灵活性是在某些

设备中简化存储器访问流水线技术。为了通知设备需要正确的数据断点检测，80x86 要求用户设定"准确的"调试标志。如果设定这些标志（2.16 中描述），则在执行下条指令前要求设备报告指令的数据断点。当然，指令执行速率还令人满意。记住断点仅在 80376，80386 和 i486 上支持。

2.2 章节组织

本章的许多节刚刚描述过，即使用指令执行循环的顶层描述做为起始点。为了完成全面的描述，则必要叙述大量的后台算法。而且，必须规定 80x86 数据结构。

由 80x86 识别的数据结构由 CPU 寄存器（也就是，通用寄存器）和 80x86 直接工作的存储器中的结构（也就是段描述符）组成。80x86 支持的基本数据类型在 2.3 中描述；而且在 2.4 中描述 80x86 寄存器组，在存储器地址生成过程中 80x86 使用的数据结构在 2.7 中描述。这一节也描述了存储器地址生成算法。除了存储器以外，80x86 结构支持 I/O 地址空间的概念。I/O 地址附属于 I/O（外围）设备。使用 I/O 空间读和写控制外设。I/O 访问在 2.9 中描述。

2.3 数据表示

既然存储器仅是一串位，则建立表示不同数据类型的规范很重要。通常，80x86 使用二进制运算表示整数数据。如果表示符号量，则使用补码概念。

存储器的最小可寻址单元称为 BYTE（字节）。一个 BYTE 由八个连续的存储位构成。因而，它可以表示在 $0\sim2^8$ 范围（也就是 256）内的一个无符号值或在$-2^7\sim+2^7-1$ 范围内的一个符号值（取补）。因而，使用 ASCII 或 EBCDIC 编码，一个 BYTE 可以表示一个文本字符。存储器中每个 BYTE 拥有一个唯一的地址。

一个 WORD（字）是两个续的 BYTEs（也就是 16 位），并且一个 DWORD 是四个连续的 BYTEs（也就是 32 位），为了寻址一个 WORD 或一个 DWORD，使用 WORD 或 DWORD 中最初的连续 BYTE。换而言之，地址 a 处的一个 WORD 占据字节地址 a 和 a+1，地址 a 处的 DWORD 占据 a 到 a+3 的字节地址。

注意，80376，80386 和 i486 可以工作在 32 位数据结构下。在这些机器上调用一个 32 位的"WORD"更有意义，可是，为了与早期概念性一致，术语"WORD"将总是认做一个 16 位量，术语"DWORD"将用来表示一个双字（32 位）量。

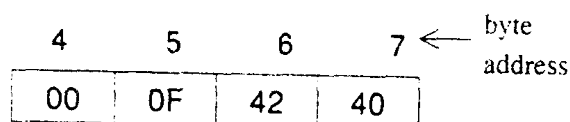
2.3.1 数据定位规则

某些结构要求一个数据项在物理存储器中自然边界上定位；也就是说，数据应该在某个地址处存储，该地址是数据类型长度的倍数。在这样的机器上，一个 WORD 仅可以在偶数字节地址处出现，一个 DWORD 在四倍字节地址处出现。在 80x86 结构中没有这样严格的限制。例如，一个 DWORD 可以在地址 5（非 4 的倍数）或地址 100（4 的倍数）处出现。不以其自然边界存储的数据项术语算做无定位。无定位数据访问处理时间长。因

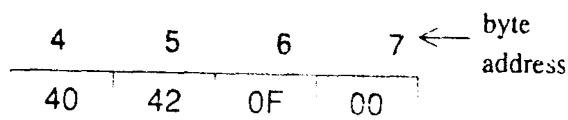
而它们应该尽可能地避免。有关在自然边界上数据定位的内容请看 2.7.8。

2.3.2 Little Endian，Big Endian 和整数表示

如果值以最小可寻址单元（也就是 BYTE）存储，这没有问题。可是，一个单数据项需要占用多个字节（也就是，一个五字符串或一个 DWORD 整数），则就会出现字节顺序问题。例如，假定在地址 4，5，6，7 以 DWORD 形式即 32 位量存储整数 1 000 000（也就是十六进制 0F4240）。最小字节（值 0x40）应该存在地址 4 还是地址 7？两种方案都可行，如图 2-1 所示。最小字节存放在最小地址数处的存储方案称做 Little Endian 方案；最小字节存放在最大地址处的方案称做 Big Endian 方案。80x86 结构使用 Little Endian 方案存储数据量。



(a) 1,000,000 in Big Endian



(b) 1,000,000 in Little Endian

图 2-1 Lig Endian 和 Little Endian 表示

Little Endian 存储方案如果存储器地址从左向右增加的，则它拥有反向存储的缺点。如果地址从右向左增加，则它改为正向，例如，0x0F4240 的反向 Little Endian 表示如图 2-2 所示。在本书中将总是使用从右向左规范表示存储器存储。

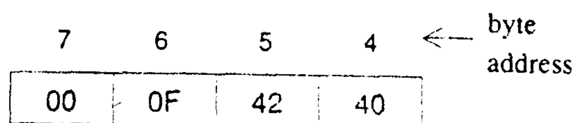


图 2-2 1 000 000 的反向 Little Endian

如果在 Big Endian 计算机上生成的数据读入一台 80x86 计算机。则字节顺序必须由软件颠倒。对于一个 WORD 数据项，XCHG 指令可用来完成这个转换，对于一个 DWORD 数据项，可以使用 BSWAP 指令（仅在 i486 上有效）。

2.3.3 表示字符串

到目前为止只讲述了整数表示。那么字符串如何表示？例如，如同一个五字符字符串"Hello"存在地址6到10中，则'H'应该存在地址6还是地址10中呢？大多数机器，无论其使用Little Endian或Big Endian方案存放整数，都在地址6处存放'H'；也就是说，字符串头个字符总是存在最小地址数处。在80x86存储器中"Hello"的存放如图2-3所示。注意，串字符在打印纸上以反向顺序出现。这是因为我们从右向左增加字节地址。这不符合字符串的习惯，但与Little Endian整数相符。

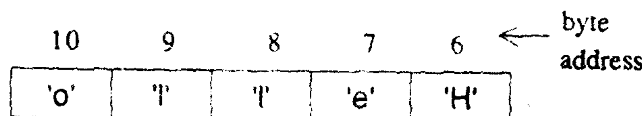


图2-3 存储器中的字符串存储

2.3.4 表示未压缩的BCD数据

有些应用宁愿以十进制而不是二进制表示数据。这可以通过以四位编码每个十进制数来完成。（也就是以四位字节）。每个四位字节存储一个0到9之间的值。10到15之间的编码不使用。这样的一个方案称做二进制码的十进制或BCD。如果每个BYTE的低四位字节用来存放一个十进制数字，则表示称为未压缩或区域格式BCD。图2-4显示了以未压缩BCD形式表示的十进制数213，600的表示方法。在这种表示中，使用ASCII字符表示数字。ASCII数字'0'到'9'对应0x30到0x39。既然数字以字符串形式存放，则以从右到左的写方案中它以倒向形式存放。

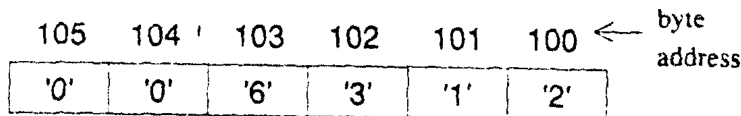


图2-4 213 600的未压缩BCD表示

80x86结构不直接支持未压缩BCD计算。它提供结合正常二进制运算指令的指令完成单个数字BCD运算。这些原函数必须重复使用才能完成通用的未压缩BCD运算。未压缩BCD支持的指令如下。

- AAA：在Add（加）后ASCII调整AL
- AAD：在Dividl（除）之前ASCII调整AX
- AAM：在Multiply（乘）后ASCII调整AX
- AAS：在Subtract（减）后ASCII调整AL

2.3.5 表示压缩的 BCD 数据

压缩的 BCD 与未压缩的 BCD 类似。可是，不同之处在于在每个字节中存放两个数字。它在未压缩 BCD 不用的高四位字节中放入了数字；高四位字节存放更高的有效数字。图 2-5 显示了以压缩 BCD 形式存放十进制值 213 600 的表示法。当以右到左方案写时读数字相当困难；每对数字以正向次序出现，而对本身以反向次序出现。例如，80x87 处理器扩展支持的 PACKED__BCD 数据类型在第一个操作数字节中最小有效低四位字节中存放最小有效数字。格式看图 3-6。

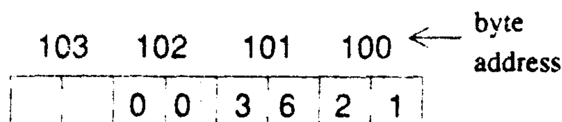


图 2-5 213 600 的压缩 BCD 表示

与不压缩 BCD 运算一样，80x86 结构不直接支持压缩 BCD 运算。而是，它提供了两条指令完成调整二进制加和减结果的工作以便它们模拟两位数字压缩十进制加和减。若完成压缩的 BCD 乘运算必须重复执行这些原始指令。这些指令是:

- DAA: 在 Add 后十进制调整 AL
- DAS: 在 Subtract 后十进制调整 AL

如果使用 80x87 浮点处理器，则压缩 BCD 和浮点间的转换指令有效。这些指令结合 80x87 提供的扩展浮点支持使用以提供压缩的 BCD 运算。更详细的内容请看第三章。

2.3.6 浮点

浮点支持不属于 80x86 结构功能的一部分。它是经 80x87 处理器扩展提供的。在除了 i486 以外所有 80x86 系列产品中，这个协处理器是一块单独的芯片。在 i486 上，80x87 指令组集成化在 i486 芯片。浮点支持是基本结构的一个扩展，它将在第三章中详细讨论。

2.4 寄存器组

本节定义了 80x86 结构支持的所有寄存器。寄存器以 C 语言说明和存储器图示的方法显示。大多数寄存器有 32 位宽。结构中 16 位的设备（即 8086，80186 和 80286）通常是被截至 16 位的。而且，某些寄存器（或它中的位字段）只在某些设备中提供，所有寄存器（和寄存器中的字段）在所有设备中不总是有效的，它们在寄存器说明中识别。在存储器图中，使用不同的背景色表示这个事实。

试读结束：需要全本请在线购买： www.ertongbook.com