原书第十章

第四部分 中央处理单元

第十二章

指令集：特征和功能

12.1机器指令特性

机器指令的元素

指令表示

指令类型

地址数目

指令集设计

12.2种操作数

数字

文字

逻辑数据

12.3英特尔x86和ARM数据类型

x86数据类型

ARM数据类型

12.4种操作

数据传输

算术

逻辑的

转换

输入输出

系统控制

控制权转移

12.5英特尔X86和ARM操作类型

x86操作类型

ARM操作类型

12.6个关键术语、复习问题和问题

附录12A小端，大端，和双端

学习目标

学习本章后，你应该能够：

概述**机器指令**的基本特征。

描述典型机器指令集中使用的操作数的类型。

概述X86和ARM数据类型。

描述典型机器指令集支持的操作数的类型。

概述X86和ARM操作类型。

理解大端（big endian）、小端（little endian）和（双端）bi-endian之间的区别。

本书中讨论的大部分内容对于计算机用户或程序员来说实际上是看不到的。如果程序员使用Pascal或者Ada等高级语言，底层机器的体系结构对其几乎不可见。

将计算机设计人员和计算机程序员衔接而又区分开的分界是机器指令集。从设计者的角度来看，机器指令集为中央处理器（CPU）提供了功能性需求：实现CPU的任务大部分都涉及机器指令集的实现。从用户的观点来看，选用机器语言（实际上是汇编语言；见附录B）进行编程的用户必定要通晓机器所直接支持的寄存器和存储器结构、数据类型以及算术逻辑单元（arithmetic and logic unit，ALU）的功能。

对计算机机器语言的描述将大大有助于了解计算机。因此在本章和下一章中将集中讨论机器指令。

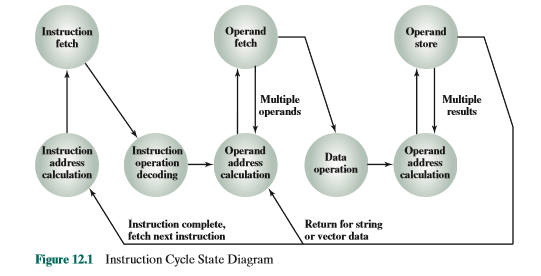
**12.1机器指令特征**

CPU的操作由它所执行的指令决定，这些指令称为机器指令（machine instruction）或计算机指令（computer instruction）。CPU能执行的不同指令的集合称为CPU的指令集（instruction set）。

**12.1.1机器指令要素**

每条机器指令必定包含处理器执行该指令所需的信息。图12-1是图3-6的复制，它表示了指令执行步骤，也隐含地定义了机器指令要素。这些要素如下所示。

**·操作码**（operation code）：指定将要完成的操作（如ADD、I/O等）。这些二进制代码常被称为**操作码**（opcode）。



**·源操作数引用**（source operand reference）：操作可能会涉及一个或多个源操作数，即操作所需的输入。

**·结果操作数引用**（result operand inference）：操作可能会产生一个结果。

**·下一条指令引用**（next instruction refrence）：告诉处理器这条指令执行完毕后到哪去取下一条指令。

待取的下一条指令的地址可能是一个实地址（real address），也可能是一个虚地址（virtual address），这取决于具体的计算机体系结构。通常来说，这个问题跟指令集无关。大多数情况下，待取的下一条指令就位于当前指令之后，此时指令中没有显示引用。当需要显式引用时，指令必须提供主存或虚拟存储器的地址。地址提供的形式在第13章讨论。

源和结果操作数可能位于如下4个范围内：

·**主存或虚存**（main or virtual memory）：与下一个指令的引用一样，必须提供主存或虚存的地址。

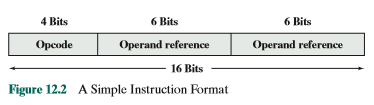
·**处理器寄存器**（processor register）：除极少数例外，处理器包含一个或多个能被机器指令所访问的寄存器。若仅存在一个寄存器，对其的引用可以是隐含的。若不仅一个寄存器，则要给每条寄存器指定唯一的名称或编号，且指令必须包含所需寄存器的编号。

·**立即数**：操作数的值直接保存在当前执行指令的某个字段中。

·**I/O设备**：需要I/O操作的指令必须指定I/O模块或设备。若使用存储器映射（memory-mapped）I/O方式，那么形式上将是另一个主存或虚存地址。

**12.1.2 指令表示**

在计算机内部，指令由一个位串表示。该指令被划分为多个字段，对应于指令的各要素。图12-2显示了一个简单的指令格式例（instruction format）子。另一个例子是示于图2-2的IAS指令格式。



大多数指令集使用不止一种格式。指令执行期间，一条指令被读入处理器中的指令寄存器（IR）中。处理器必须能够从各种指令字段中提取数据来完成所需操作。

无论是编程人员还是相关图书的读者都难与处理机器指令的二进制表示法直接打交道，故普遍使用机器指令符号表示法（symbolic representation）。在表1-1中的IAS指令集使用了一个示例。

操作码被缩写成指示操作的助记符（mnemonic）。常见的例子包括：

ADD 加

SUB 减

MUL 乘

DIV 除

LOAD 由存储器载入

STOR 保存到存储器

操作数也可以用符号表示。例如，指令：

ADD R,Y

会意味着将存储器Y中的数据值加到寄存器。在这个例子中，Y是存储器中某个位置的地址，而R指的是一个特定的寄存器。注意，操作是对位置的内容执行的，而不是对其地址。

因此可以用符号形式编写机器语言程序。每条符号化的操作码都有一个固定的二进制表示，程序员指明每条符号操作数的位置。例如，程序员可以在程序开始处写出下列定义：

X = 513

Y = 514

等等。一个简单的程序会专门接受这些符号输入，将操作码和操作数引用转换成对应二进制形式，并构成二进制机器指令。

机器语言程序员已经稀少到几乎没有的程度。今天大多数程序都是用高级语言编写的，或至少也是以汇编语言（附录B将讨论）来编写的。然而符号机器语言仍然是描述机器指令的有力工具，因此我们还将使用它来学习有关机器指令的知识。

**指令类型**

考虑像能被BASIC或FORTRAN等语言表达的高级语言指令。例如：

X = X + Y

这条指令指挥计算机将存于Y的值加到存于X的值并将结果放入X中。如何用机器指令来完成呢？让我们假设变量X和Y对应于位置513和514。若我们假定有一简单指令集，这个操作能以如下三条指令来完成：

1. 将存储器位置513的内容装入一个寄存器；
2. 把存储器位置514的内容与上述寄存器的内容相加并保存到该寄存器中；
3. 将此寄存器内容存入存储器的513位置中。

可以看出，一条单一的BASIC指令可能需要三条机器指令。这是高级语言和机器语言之间关系的典型特征。高级语言使用变量以简明的代数形式去表示操作。而机器语言是以涉及数据移入移出寄存器的基本形式来表达操作。

按照这个简单例子给我们的启示，让我们考虑一个具体的计算机中必须包含的指令类型。计算机应该有能允许用户表达任何数据处理任务的一组指令。另一方面是考虑高级编程语言的能力。任何用高级语言编写的程序都必须转换成机器语言才能执行。于是机器指令集必须充分，足以表达来任何高级语言的指令形式。基于这些考虑，可将指令类型分类成：

**·数据处理**：算术和逻辑指令。

**·数据存储**：存储器指令。

**·数据移动**：I/O指令。

**·控制**：测试和分支（branch）指令。

算术指令提供了处理数值型数据的计算能力。逻辑（布尔）指令是对字中的位进行操作，这些位不再看成是数值位。因此提供了处理用户可能希望使用的任何其他类型的数据的能力。这些操作主要对处理器寄存器中的数据执行。因此，必须有内存指令用于在存储器和寄存器之间传送数据。需要有I/O指令将程序和数据装载到存储器，并将计算结果返给用户。测试指令用于测试数据字的值或计算的状态。分支指令则用于依据判定条件是否成立，转移到另一组指令上去。

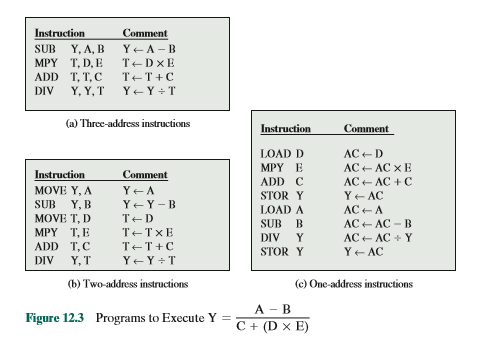
  本章稍后将更详细地考察各类指令。

**地址数目**

描述处理器体系结构的一种传统方式之一是依据每条指令包含的地址数。随着处理器设计复杂度的增长，这种量度逐渐失去了意义。然而对于规划和分析来说还是有用的。

一条指令需要的最大地址数是多少？很显然算术和逻辑指令要求的操作数最多。实际上所有的算术和逻辑运算要么是一元的（一个源操作数），要么是二元的（两个源操作数）。因此，我们最多需要两个地址来访问源操作数。运算的结果必须存储，意味着需要第三个地址，用以定义目的操作数（destination operand）。最后，在完成一条指令之后必须获取下一个指令，这又需要指令地址。

上述推过程表明，一条指令可能需要有四个地址引用：两个源操作数、一个目标操作数和下一条指令地址。实际上在大多数系统中，指令使用一个、两个或三个操作数地址，而下一指令地址为隐含的（从程序计数器获得）。很多系统包含一些需要更多操作数的特殊指令。例如，在第13章所描述的ARM处理器，其装载和保存指令组可以在单个指令中使用多达17个寄存器操作数。

图12-3比较了可用于计算Y = ( A – B ) / [ C + ( D × E ) ]的典型的单地址、双地址和三地址的指令。使用三地址，每条指令可指定两个源操作数位置和一个目的操作数位置。因为我们希望不更改任何操作数位置的值，故使用T为中间结果的暂存。注意，原始的表达式有5个操作数，而使用三地址只需要4条指令。

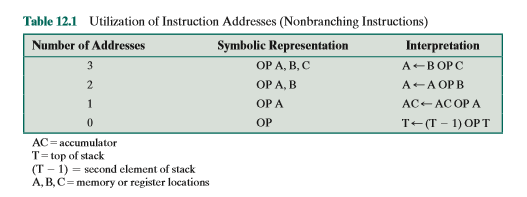
三地址指令格式并不普遍，因为指令格式要容纳三个地址引用，所以指令格式会相对较长。如果采用双地址指令完成一次二元运算，那么一个地址必须既用作源操作数又用作结果，执行双重任务。于是，SUB Y，B指令执行Y – B计算，并将结果保存于Y。双地址格式降低了空间需求，但也引入了一些麻烦。为了避免更改一个操作数的值，要使用MOVE指令在完成运算之前将一个值传到最终或临时位置上去。图12-3给出的示例中，使用两地址指令的话需要6条指令。

最简单的还是地址指令。为使它能工作为此，第二个地址必须是隐含的。这在早期的机器中是很普遍的，其隐含的地址是被称为**累加器**（Accumulator，AC）的CPU寄存器。累加器提供其中一个操作数，且结果被保存回累加器。图12-3的例子显示，完成同样的任务需要8条单地址指令。

事实上，对于某些指令还可能有“零地址”格式。零地址可用于称为栈（stack）的专门存储器中。栈是一种后入先出的结构，位于某个已知的位置，并且通常栈顶至少是放在CPU寄存器中。于是零地址指令能访问栈顶两个元素。附录I中给出了栈的描述，本章稍后以及第13章将进一步说明栈的使用。

表12-1总结了具有0、1、2、或3个地址的指令的解释。表中的每种情况都假定了下一指令的地址是隐含的，指令的运算都是对两个源操作数进行，并产生一个结果操作数。

每条指令的地址数目是基本的设计决策。每条指令中的地址数目越少，指令的长度就越短，指令也更原始（不需要复杂的CPU）。另一方面，它又会使程序总的指令条数更多，这通常导致更长的执行时间和更长更复杂的程序。此外，在单地址指令和多地址指令之间存在一个重要分界点。对单地址指令来说，程序员通常只有一个通用寄存器（即累加器）可利用。而对于多地址指令，普遍具有多个通用寄存器可用。这就允许某些计算操作只使用寄存器即可完成。因为访问寄存器比访问内存快得多，可以加快执行速度。为了灵活性和能使用多个寄存器，大多数当代计算机采用了双地址和三地址指令的混合方式。



涉及每条指令地址数目选择的设计权衡被其他因素复杂化了。一个需要考虑的设计问题是引用到存储器还是寄存器。因为寄存器的数目总是不太多，所以寄存器引用所需的位更少。而且，正如下一章即将讲述的内容，一个机器会有各种寻址方式，为指定这些方式也要占用指令的一位或几位。结果大多数处理器被设计成有多种指令格式。

**指令集设计**

计算机设计中最有趣也最受关注的部分是指令集的设计。指令集的设计非常复杂，因为它影响着计算机系统的诸多方面。指令集定义了处理器应完成的多数功能，因此对处理器的实现有着显著的影响。指令集也是程序员控制处理器的手段。于是在设计指令集时必须考虑程序员的要求。

当你知道，在有关指令集设计的最根本出发点方面还存在某些争议时，你一定会感到惊奇。事实确实如此。近年来这种分歧还在不断加大。这些最基础的设计出发点中，最重要的包括：

**·操作指令表**（operation repertoire）：应提供多少和什么样的操作，操作将是何等复杂。

**·数据类型**（data type）：对哪几种数据类型完成操作。

**·指令格式**（instruction format）：指令的（位）长度、地址数目、各个字段的大小等。

**·寄存器**（register）：能被指令访问的处理区寄存器数目以及它们的用途。

**·寻址**（addressing）：指定操作数地址的产生方式。

这些出发点是高度相关的，设计指令集时必须一起考虑。当然本书只能依次考察它们，但我们也努力说明其相关性。

由于这个主题的重要性，第三部分的许多篇幅将用于介绍指令集设计。在此节综述之后，本章将考察数据类型和操作清单。第13章考察寻址模式（其中也包括寄存器的考虑）和指令格式。第15章考察精简指令集计算机（RISC）。RISC体系结构对很多商业计算机中指令集设计的传统方式提出了质疑。

**12.2 操作数类型**

机器指令对数据进行操作。数据的最重要的一般类别是：地址、数值、字符、逻辑数据。

在第13章讨论寻址方式时将看到，地址实际也是一种形式的数据。多数情况下，必须对指令中的操作数引用来完成某些计算，才能以确定主存或虚拟地址。此时地址将被看成是无符号整数。

  其他普通数据类型是数值、字符和逻辑数据，本节将分别予以简要的考察。除此之外，某些机器还定义了专门的数据类型或数据结构。例如，有些机器的指令可对字符列表或字符串直接操作。

**数值**

所有机器语言都包括数值型数据类型。即使在一个非数值数据的处理中，也需要某些数值用作计数器、字段宽度。一般数学所用的数值与计算机中存储的数值之间存在一个重大区别是后者是有限的。这表现在两个方面：首先是机器可表示数值的幅值是有限的，其次是在浮点数的情况下数值精度也是有限的。于是，程序员必须了解舍入、上溢和下溢的意义。

  计算机中普遍使用三种类型的数值数据：二进制整数或定点数、二进制浮点数、十进制数。

  我们已在第10章较详细地考察过前两种类型的数值数据。这里有必要介绍一下十进制数。

虽然计算机内部操作都有二进制属性，但作为系统的使用者——人，却是与十进制打交道。因此有必要在输入端将十进制转换为二进制，在输出端将二进制转换为十进制。对于有大量I/O需处而计算相对较少、较简单的应用来说，以十进制的形式来存储和操作更为合适。为此，一个最普遍的表示法是压缩的或称打包的十进制数（packed decimal）。①[[1]](#footnote-1)

压缩的十进制数用4位二进制代码表示每个十进制数字。0 ＝ 0000, 1 ＝ 0001，…，8 ＝ 1000，和9 ＝ 1001。注意，这是一个相当低效的代码，4位二进制值可有16个不同的值，但只上述10个代码是有效的，另外6个是无效的。为构成一个数，4位代码紧密排列在一起，通常为多个8位的字。例如，246的代码是0000 0010 0100 0110。显然，这种表示法没有二进制表示法那样紧凑但它避免了转换开销。负数可以通过在整个压缩十进制数字串的前后加上一个4位的符号数字来表示。标准的符号数字值是1100表示正数（+），1101表示负数（-）。

多数机器都提供了直接对压缩十进制数进行操作的算术指令，算法类似于9.3节的介绍，但是必须留意十进制的进位操作。

**字符**

另一种常用的数据类型是文本或字符串。虽然文本数据对人类来说使用最方便，但在数据处理和通信系统中却不便于一字符形式存储或发送。因为数据处理和通信系统都是被设计处理二进制数据的。于是，研究人员研制了几种编码方式将字符表示成二进制的位序列。最早的编码例子或许是莫尔斯电码（Morse Code）。今天使用最广泛的字符编码是国际参考字母表（International Reference Alphabet，IRA），在美国称为ASCII码（美国信息交换标准代码，American Standard Code for Information Exchange），见附录H。在这种编码中，每个字符被表示成唯一的7位二进制串，于是共有128个可表示字符。这个数量比可打印字符数量多，故某些“位样式”（Bit Pattern）用来表示“控制字符”（control character）一些控制字符用来控制字符的按页打印；而另一些则用来控制通信过程。IRA编码的字符几乎总是以每字符8位来存储和发送的。第8位可以设置为0或用作用于错误检测的奇偶校验位。后一种情况下，位要被设置为使得在每个8位中二进制1的数目总是奇数个(奇校验)或总是偶数个(偶校验)。

注意，在表H-1（附录H）中数字0~9的IRA代码位样式是011XXXX，其中最后4位XXXX恰是0000到1001的表示，即压缩十进制数的编码。因此在7位IRA代码和4位压缩十进制表示之间转换时非常方便的。

用于字符编码的另一种代码是EBCDIC（Extended Binary Coded Decimal Interchange Code扩展二进制编码十进制交换码）。它是一种被用于IBM制造的大型机种的8位编码。与IRA一样，EBCDIC与压缩十进制是兼容的。EBCDIC代码的11110000到111111001表示数字0到9。

**逻辑数据**

正常情况下，每个字或其他可寻址单元（字节、半字等）是作为一个单一数据单元看待的。然而某些时候需要讲一个n位单元看成是由n个1位项组成，每项有值0或1。当以这种方式看待数据时，该数据就被认为是逻辑数据。

这种位排列观点有两个优点。第一，有时我们希望存储一个布尔或二进制数据项序列，序列中的每项只能取值1（真）和0（假）。对于这种情况，逻辑数据能实现存储器最有效的使用。第二，逻辑数据有利于实现对数据项的具体位进行操纵。例如，如果浮点运算是以软件实现的，那么我们需要能够在某些操作中移动有效位。另一个例子是，由IRA转换到压缩十进制时，我们需要提取出每字节的最右边4位。

注意，在上例子中中，同一个数据有时被看作是逻辑数据，有时则被当作数值或文本。数据单元的类型由当前在它上面正在完成的操作所确定。虽然在高级语言中（如Pascal语言）通常不是这种情况，但在机器语言中几乎总是如此。

**12.3 Intel x86和ARM数据类型**

**x86数据类型**

x86能处理8位（字节）、16位、32位（双字）、64位（四字）和128位（双四字）各种长度的数据类型。为允许最大的数据结构灵活性和最有效地使用存储器，字不需要在偶数地址上对齐，双字也不需要在4倍整数地址上对齐，四字也不需要在8倍整数地址上对其，其他类推。然而当经由32位总线存储数据时，数据传送是以双字为单位进行的，双字的起始地址是能被4整除的、处理器要将对于未对齐值的访问请求，转换成一序列的总线传送请求。像所有的80x86机器一样，x86也采用小端序（little-endian ordering）风格，即最低有效字节存于最低地址中（见本章后面的附录12A对端序的讨论。）

字节、字、双字、四字和双四字被称为常规数据类型。另外，x86支持一系列的特殊数据类型，这些特殊数据类型只被特殊指令所识别和操作。表12-2总结了所有这些数据类型。

图12-4展示了x86数值数据类型。有符号整数是以2的补码表示的，可以是16、32或64位长。浮点数类型实际指可被浮点运算单元使用和可被浮点指令操作的一组数据类型作。三种浮点表示都是符合IEEE 754标准的。

压缩的SIMD（single-instruction-multiple-data,单指令多数据）数据类型是为了优化多媒体应用性能，作为一种指令集的扩展，引入x86体系结构中来的。这些扩展包括MMX（multimedia extension，多媒体扩展）和SSE（streaming SIMD extension，流式SIMD扩展）。基本思想是把多个操作数打包为一个内存引用项，从而提高性能。压缩的SIMD数据类型包括：

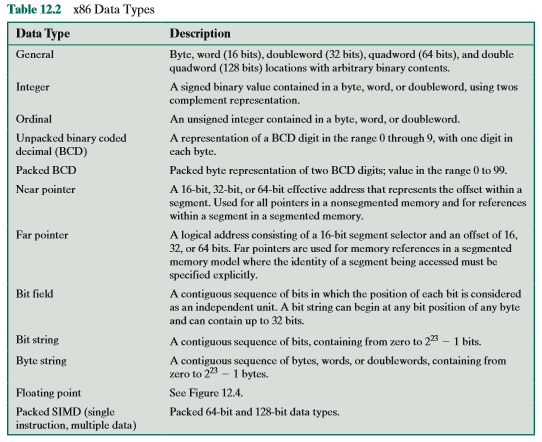
**·压缩的字节和压缩的字节整数：**多个字节被打包到一个64位的四字（quadword）或128位的双四字（double quadword）中，并被当作位字段或整数。

**·压缩的字和压缩的字整数：**多个16位的字被打包到一个64位的四字或128位的双四字中，并被当作位字段或整数。

**·压缩的双字和压缩的双字整数：**多个32位的字被打包到一个64位的四字或128位的双四字中，并被当作位字段或整数。

**·压缩的四字和压缩的四字整数：**两个64位的字被打包到一个128位的双四字，并被当作位字段或整数。

**·压缩的单精度浮点数和压缩的双精度浮点数：**四个32位的单精度浮点数或两个64位的双精度浮点数被打包到一个128位的双四字中。



**ARM数据类型**

ARM处理器支持8位（字节）、16位（半字）、32位（字）各种长度的数据类型。通常，半字数据的访问要对齐到半字地址，字数据的访问要对齐到字地址。对于未对齐的访问请求，ARM提供了3种可选方案。

**·默认情况：**

非对齐的地址将被截断，即当访问字时，地址位[1:0]会被当作0，而访问半字时，地址位[0]被当作0。

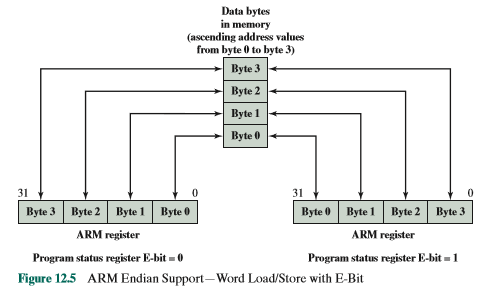
载入单个字的ARM指令遇到非字对齐的地址时，会以此地址对字对齐的数据循环右移一个、两个或三个字节，具体操作根据非对齐地址最低两位的值而定。

**·对齐检查：**如果设置了相应的控制位，在试图访问非对齐地址时，会产生一个数据取消（data abort）信号，表明发生了一个非对齐访问错误。

**·非对齐访问**：如果设置了启用此选项时，处理器将通过一次或多次内存访问，获得非对齐地址所指定的字节，并返回给程序。

所有这三种数据类型（字节、半字和字）都支持使用无符号表示，此时数据所表示的值是一个无符号、非负的整数。所有数据类型也支持使用2的补码表示有符号整数。

大部分ARM处理器实现都不支持浮点硬件，这可以节省功耗和芯片面积。如果这些处理器需要浮点运算，那么就只能用软件实现。ARM确实提供一个浮点协处理器，利用浮点协处理器可以支持IEEE 754所规定的单精度和双精度浮点数据类型。



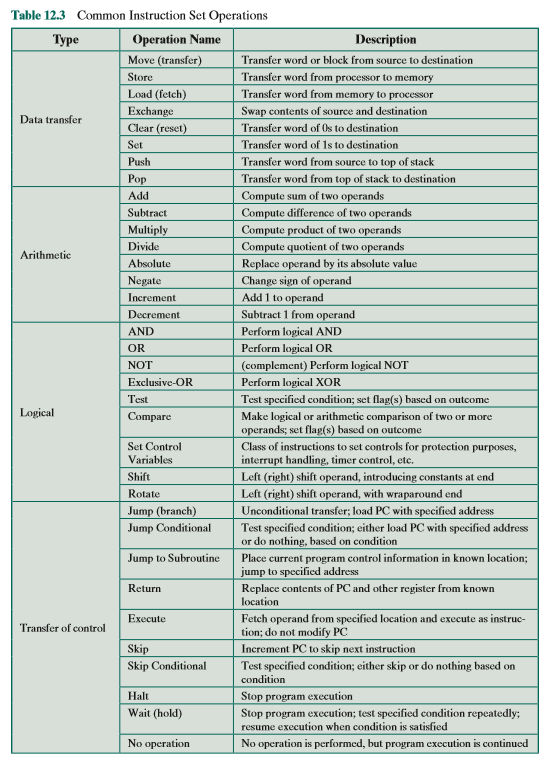
**端序支持**

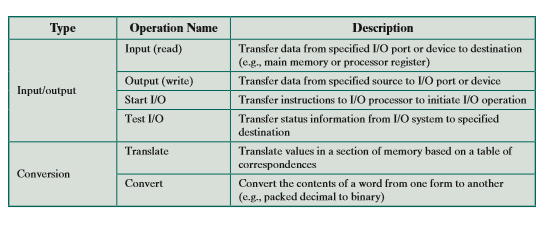
程序可以通过SETEND指令设置和清除ARM处理器中系统控制寄存器的端序状态位（E位），E位定义了采用哪种端序来装载保存数据。图12-通过字装载和保存操作显示了E位的功能。当设计人员需要使用不同端序来访问操作系统和环境系统数据结构时，端序选择机制位系统设计人员在动态装载和和保存数据时提供了方便。注意，每个数据字节的地址在内存中是固定的，不过寄存器中的字节排列在不同端序下是不同的。

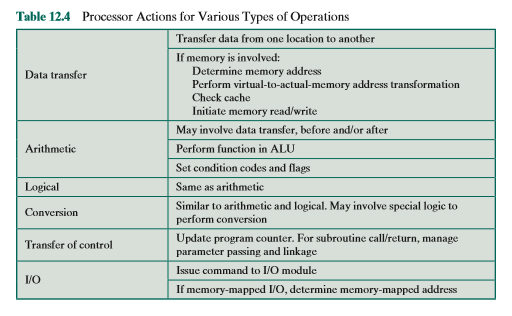
**12.4操作类型**

不同操作码的数量因机器而异。然而，在所有机器上都会发现相同的常用操作类型。操作的典型分类如下：数据传送、算术、逻辑、转换（Conversion）、输入/输出、系统控制、控制转移。

表12-3（基于[HAYE98]）列出了每一类操作的常见指令类型。本节简要介绍这些不同类型的操作，并结合处理器为执行具体操作类型时所采取的动作（总结见表12-4中）。后一主题还要在第14章中进行更详细介绍。





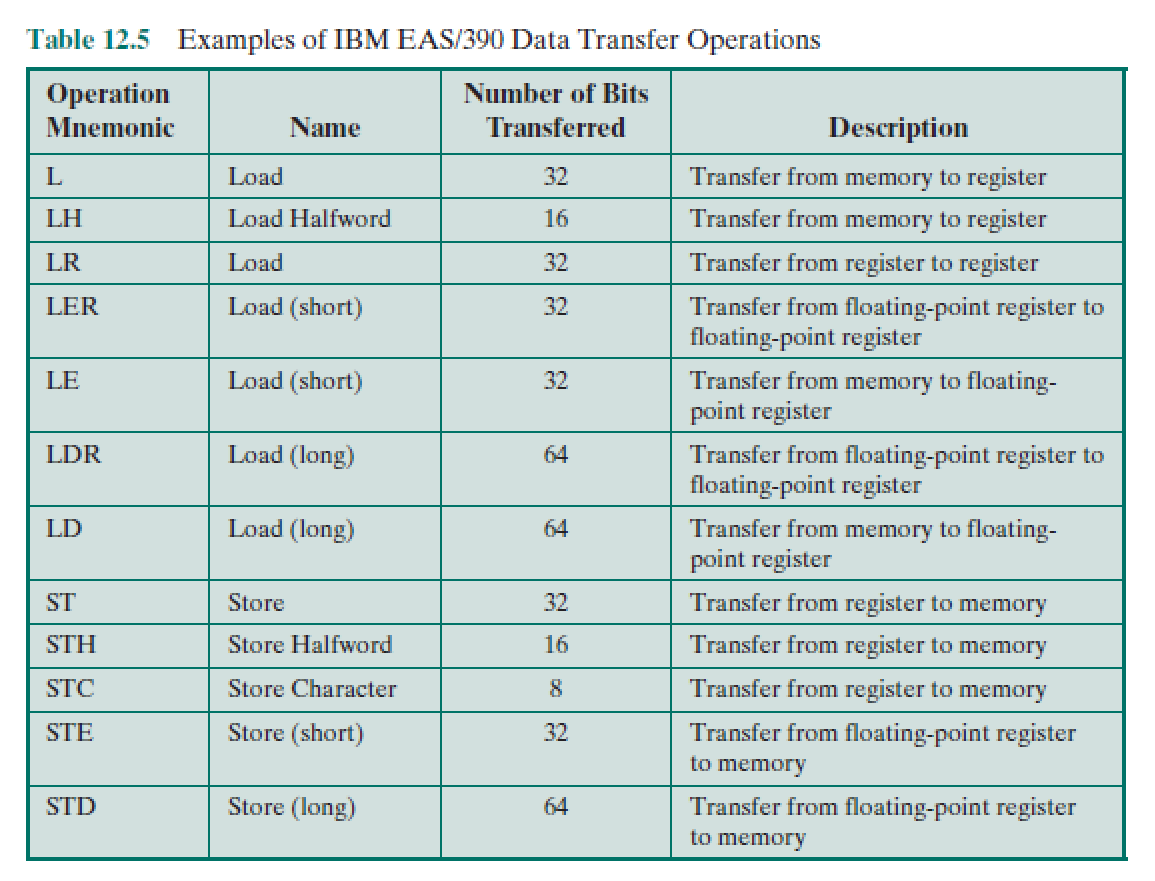


**数据传送**

最基础的机器指令类型是数据传输指令。数据传输指令必须指明几件事情。第一，源和目标操作数的位置。每个位置可以是存储器、寄存器或栈顶。第二，必须指明要传送数据的长度。第三，像所有带操作数的指令一样，必须位每个操作数指明寻址方式。最后一点将在第13章讨论。

选择要包含在指令集中的数据传输指令执行了设计者必须做出的各种权衡。例如，操作数的一般位置（内存或寄存器）可以在操作码的具体说明或操作数中指示。表12.5显示了最常见的IBMIAS/390数据传输指令的示例。请注意，存在变量来指示

指令集内选取什么样的数据传送指令，是设计人员必须权衡考虑的事情的一个范例。例如，操作数的通常为止（存储器或寄存器）是以操作数或操作码的类型来指明。表12-5列出了最常用的IBM S/390的数据传送指令例子。注意，这里存在变量用于表示将要传送数据长度（8/16/32或64位）。另外，对寄存器到寄存器，寄存器到存储器和存储器到寄存器的传送，采用的是不同的指令。与之对比，VAX机是使用带有不同传送长度变量的MOV指令，不过它把指示操作数是在寄存器还是在存储器中的信息，作为操作数的一部分。VAX的方法对于程序员来说是相对容易的，只需与较少的助记符打交道。然而它没有IBM S/370方法那样紧凑，因为每个操作数的位置（寄存器还是存储器）必须在指令中分别指定。当下一章讨论指令格式时我们再返回到这个问题上来。



就处理器动作而言，数据传送操作也许是最简单的类型。若源和目标都是寄存器，则处理器只要使数据从一个寄存器传送到另一个即可；这是处理器内部操作。若一个或两个操作数是在存储器中，则处理器必须完成如下的某些乃至全部动作：

1. 根据寻址方式（第13章将讨论），计算存储地址。
2. 若地址指向虚拟存储器，则将虚地址转换成物理存储地址。
3. 确定所寻找的项是否在高速缓存（cache）中。
4. 如果不在，向存储器模块发命令。

**算术运算**

大多数机器都提供了加、减、乘、除这样的基本算术指令。这些操作总是为有符号整数（定点数）提供，也经常为浮点数和压缩十进制数提供。

其他可能的操作包括各种单操作数指令。例如：

·Absolute：取一个操作数的绝对值。

·Negate：取一操作数的负数。

·Increment：操作数加1。

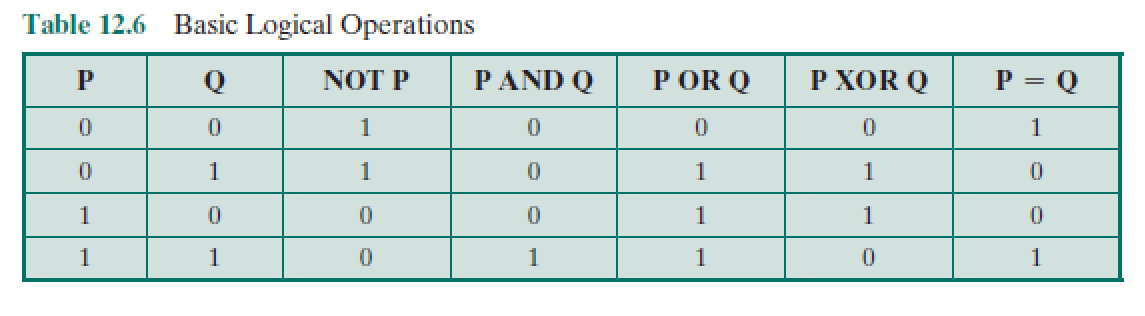
·Decrement：操作数减1。

一条算术指令的执行会涉及数据传送操作，来为算术和逻辑单元（ALU）准备输入，并传送ALU的输出。图3-5说明了在算术运算中所涉及的数据传送活动。当然，处理器的ALU部分还要完成所要求的运算。

**逻辑运算**

大多数机器也提供处理一个字或其他可寻址单元中的个别位的操作，这常被称为“位操纵”（bit twiddling）。这些操作的基础是布尔运算（见第11章）。

能在布尔或二进制数据上完成的某些基本逻辑操作列于表12-6中。NOT操作取一位的反。与或、异或（XOR）是有两个操作数的最常见逻辑功能。EQUAL是一个有用的二进制测试。



这些逻辑操作能施加到n位的逻辑数据单元上。于是，若两个寄存器含有如下数据

（R1）＝10100101

（R2）＝00001111

则

(R1) AND (R2) = 00000101

其中（X）表示位置X中的内容。从这个例子可以得出，AND操作能用来屏蔽（mask）一个字，选出字中的某些位，而其他位为0。作为一个例子，若两个寄存器含有：

（R1）＝10100101

（R2）＝11111111

则

（R1）XOR（R2）＝01011010

对于一个全1的字，XOR操作则会将另一字的各位取反（1的补码）。

除了按位的逻辑操作，大多数机器也提供了移位和旋转（rotating）的功能。最基本的操作说明在图12-6中。使用**逻辑移位**（logical shift）可以让一个字的各位左移或右移，一端移出的位丢失，另一端则是移入0。逻辑移位主要用于隔离字中的各位段。移入字中的那些0挤走了不需要的信息，它们从另一端被移去。

举个例子，假设我们希望每次1个字符地将数据字符发送到I/O设备。若每个存储器字是16位长且含有两个字符，则在发送之前必须先拆包字符。送出字中的两个字符需要：

1. 将此字装入一个寄存器。
2. 右移8次，将剩下的字符移到寄存器的右半部。
3. 执行I/O操作。此时I/O模块将读取数据总线的低八位。

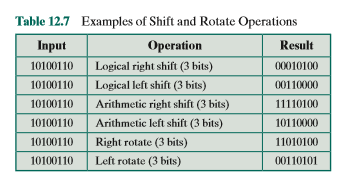
上述步骤发送了该字左半部的字符，要发送右半部的字符：

1. 再一次将此字装入寄存器。
2. 和0000000011111111进行与（AND）操作。这将屏蔽掉左半部的字符。
3. 执行I/O操作。

**算术移位**（arithmetic shift）操作是将数据看作有符号整数而不移符号位。对于算术右移，符号位一般是复制到它右边空出的位。对于算术左移，除了符号位不变外，其余位的操作和逻辑左移一样。这些操作能加速某些算术运算。对于以2的补码表示的数，算术右移相当于除以2；原数为奇数的话，则被截断。无溢出时，算术左移和逻辑左移都相当于乘以2。因为有潜在的溢出问题，许多处理器，包括PowerPC和Itanium，不提供这种左移指令，其他的比如IBM EAS/390，提供这种指令。奇怪的是，x86体系提供了一条算数左移指令，但将它定义成等同于逻辑左移指令。

**旋转**（rotate）或**循环移位**（cyclic shift）操作保留了被操作数的所有位。旋转的一个用途是连续地将各个位移到最左位，从而可通过测试符号位（将其当作一个数）来识别各个位。

与算术操作一样，逻辑操作涉及ALU动作并且会涉及数据传送操作。表12-7给出了本小节讨论过的所有移位和旋转操作的例子。



**转换**

转换指令改变数据格式或对数据格式进行操作，例如从十进制转到二进制。一种更复杂的编辑转换指令如EAS/390的翻译（Translate，TR）指令。这条指令能用来将一种8位编码转换成另一种形式编码，它有三个操作数：

TR R1（L），R2

R2包含一个8位编码表的起始地址。指令以R1指定地址开始，翻译连续的L个字节，每个字节被以此字节位索引的编码表项内容所替代。例如，由EBCDIC转换成IRA。我们首先要在存储位置，例如1000~10FF的十六进制值，生成一个256字节的表。表项的内容是字符的IRA编码，表项的排序是按EBCDIC编码的二进制表示值的大小，由小到大排列；即某字符的IRA编码放入表中的相对位置等于此字符的EBCDIC编码值。例如，数字0到9的IRA编码值为30到39，其EBCDIC编码值位F0到F9，故在10F0到10F9的表项位置中有值30到39。现在假定在2100位置处开始有数字1984的EBCDIC码。我们要求把它们转换成IRA编码。假定：

·位置2100~2103处含有F1 F9 F8 F4

·R1中含2100

·R2中有1000

若我们执行：

TR R1（4），R2

位置2100~2103将会有31 39 38 34。

**输入/输出**

第7章已经较详细地讨论了输入/输出（I/O）指令。正如我们看到的，输入/输出可选取几种方式，包括分立的编程控制的I/O、存储映射的编程控制的I/O、DMA和使用I/O处理器。多数实现是只提供少数几条I/O指令，由参数、代码或命令字来指定所要求的动作。

**系统控制**

系统控制指令通常是特权指令，仅当处理器正处于某种特权状态时，或程序正在一个专门的特权存储区域中执行时才能被执行。一般而言，这些系统指令保留给操作系统使用。

系统控制操作的某些例子如下，一条系统控制指令可读取或更改控制寄存器的内容，我们将在第14章讨论控制寄存器。另一个例子是在EAS/390存储系统中使用的读或修改一个存储保护键的指令。再一个例子是多道程序系统中存取进程控制块。

**控制转移**

至此所讨论过的所有操作类型，指令指定的是将要完成的操作和操作数。隐含地，下一条将要执行的指令，是在存储器中当前指令之后的那条指令。然而任何程序都有相当一部分指令具有改变指令执行顺序的功能。对于这些指令，CPU所完成的操作是将程序计数器修改为某条指令的存储器地址。

有几个理由说明为何需要控制转移操作。其中最重要的是：

（1）在计算机的实际应用中，必须能偶不止一次甚至上千次地重复执行某些指令，这种能力是至关重要的。实现一个应用需要上千条甚至上百万条指令。若每条指令必须分别写出是不可思议的。若一个表或列表需要被处理，则可使用程序循环的方法。一个指令序列重复执行直到所有数据被处理完毕。

（2）实际上，所有程序都涉及某种裁决。我们希望计算机能在满足某种条件下做某些事情，另一种条件满足时做另一种事情。例如，一个指令序列计算数的平方根。在序列开始时应该测试数的符号位，若是负数，则不进行计算而报告出错。

（3）正确地编写一个大型的即使是中等规模的计算机程序，也是一个极其困难的任务。若能将此任务分成小的片段，每次只专注于这一段的任务，会有很大益处。

现在可以开始指令集中最常见的控制转移操作的讨论了，它们是：分支（branch）；跳步（skip）；过程调用。

1. **分支指令**

分支（Branch）指令亦称为跳转（Jump）指令，它把将要执行的下一条指令的地址作为它的一个操作数。使用最多的是**条件分支**（conditional branch）指令，即仅当某种条件被满足时才进行转移（将程序计数器的值更改为操作数指定的地址）；否则顺序执行下一条指令（像通常那样程序计数器加1）。**无条件分支**（unconditional branch）指令则总是转移。 有两种常用的方式来生成条件分支指令中将要测试的条件。首先，大多数机器提供了一位或多位的条件码，它作为某种操作的结果被设置。这些条件码可以想象成一个用户可见的短寄存器。例如，算术运算（加法、减法等等）可能以4种值（0、正、负、溢出）来相应设置2位条件代码。在这样的机器上，可能有如下4种不同的条件转移指令；

BRP X 若结果是正，则转移到X位置

BRN X 若结果是负，则转移到X位置

BRZ X 若结果是零，则转移到X位置

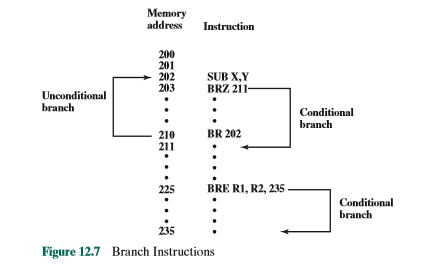
BRO X 若结果出现上溢，则转移到X位置

在所有这些情况中，设置条件码的操作结果指的是最近一次操作的结果。

另一种方法是在同一条指令内完成比较和转移，这可用于三地址指令格式的指令。例如：

BRE R1，R2，X 若R1内容=R2内容，则转移到X

图12-7给出了这些操作的例子。注意，转移可以是向前（forward，具有更高地址的指令），也可以是向后（backward，较低地址）。例子中显示了无条件和条件分支指令能用来产生一个重复的指令循环。位置202到210的指令将重复执行，直到X减Y的结果是0。



1. **跳步指令**

另一类常用的控制转移指令是跳步（Skip）指令。跳步指令包括一个隐含地址。一般跳步指令是指下一指令将被跳过；于是隐含地址等于下一指令地址加上该指令长度之和。

因为跳步指令不要求目标地址字段，所以做其他事情更有自由。一个典型的例子是“加1并且若为0则跳步”（increment-and-skip-if-zero，ISZ）指令。考虑如下的程序片段：

301

：

309 ISZ R1

310 BR 301

311

在这个程序片段中，使用了两条控制转移指令来实现一个重复循环。初始是R1设置成欲重复次数的负数。在循环终端，R1被加1。若它不是0，则程序向后转移到循环开始处，循环体再次被执行。否则，此转移指令被跳过，接续执行循环之后的指令。

1. **过程调用指令**

在编程语言的发展历程中，也许最重要的革新就是过程（Procedure）了。过程是一个自包含（self-contained）的计算机程序，能被并入到一个大的程序中。可在程序的任一点上激活或调用过程，即在那一点上，处理器被指示转到过程起始处，执行完整个过程，然后再返回到调用发生点。

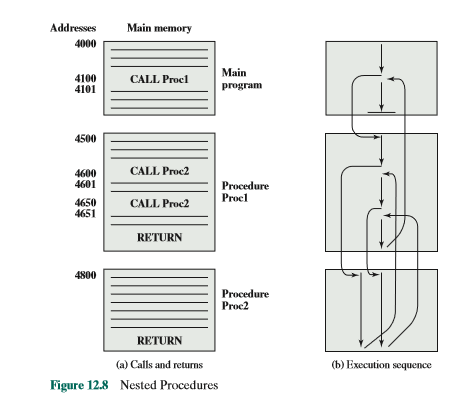
使用过程的两个基本理由是经济性和模块化。过程允许多次使用同一代码片段，这能大大节省编程工作量，并且也使系统的存储空间得到最有效的利用（程序必须被存储）。过程亦允许大的编程任务被分解成较小的任务。这种模块化方式极大地减轻了编程任务。

过程机制涉及两类基本指令：由目前位置转移到过程的调用指令；由过程返回到调用发生位置的返回指令。两者都是转移指令形式。

图12-8a说明了如何使用过程来构造程序。此例中，有一个起始位置为4000的主程序。这个主程序包括了一条调用Proc1过程的指令，Proc1的起始位置为4500。当遇到这条调用的指令时，CPU挂起主程序的执行，并由位置4500处取出下一条指令，开始Proc1的执行。在Proc1内，有两次对位于4800开始的Proc2的调用。每次Proc1被挂起而Proc2被执行。返回（Return）指令使CPU返回到调用程序，并接续执行相应调用（Call）指令之后的指令。图12-8b显示了上述的执行顺序。

有几点是值得注意的：

1. 可从多个位置调用过程。
2. 过程中能出现过程的调用，并允许过程嵌套（nesting）到任意深度。
3. 每一次过程调用与被调用程序中的一次返回相匹配。



因为能从不同位置点调用过程，所以处理器必须以某种方式保存返回地址，以使返回能相应发生。有三个常用的保存返回地址的位置：寄存器、被调过程开始处、栈顶部。

考虑一条机器语言指令CALL X，它代表调用X处的过程。若使用寄存器方法，则CALL X指令引起如下动作：

RN←PC+△

PC←X

这里的RN是用于此目的的寄存器。PC是程序计数器，△是指令长度。被调用的过程需要保存RN的内容，以便稍后的返回使用。

第二种可能的方法是将返回地址存于过程开始处。这种情况下，CALL X引起：

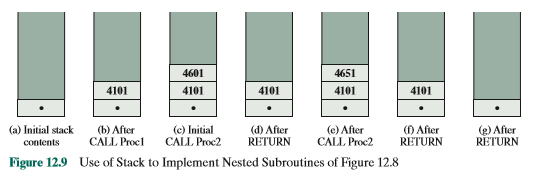
X←PC+△

PC←X+1

这是很方便的，返回地址被很安全地保存。

前两种方法都能工作并已被采用。这些方法的唯一限制是使**重入**（reentrant）过程的使用变得复杂。**重入过程**是这样一种过程，它准许几个对它的调用同时存在。递归过程（自己调用自己的过程）就是使用这一特征的例子（见附录M）。如果重入过程的参数通过寄存器或内存来传递，必须有代码负责保存参数，这样这些寄存器和内存空间就可以被其他过程调用使用了。

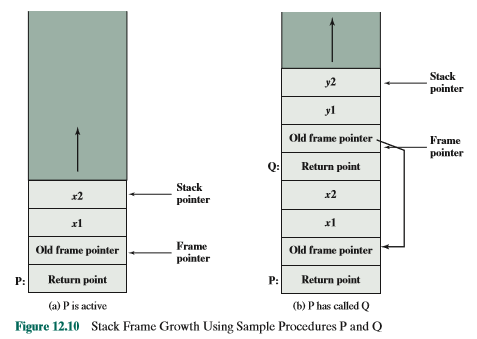
一个更通用更强有力的方法是使用栈（栈的定义见附录I）。当CPU执行一次调用时，它将返回地址放到栈上。当它执行一次返回时，它使用栈上的地址。图12-9说明了栈的使用。



过程调用除需要提供返回地址外，经常还需要传送参数。可以使用寄存器传送参数，也可以将参数存入恰好在调用指令之后的存储器位置中；则此时返回必须是返回到这些参数之后的位置上。这两种方法都有缺点。若使用寄存器，则主调用程序和被调用程序都要小心编写，以使寄存器恰当地被使用。将参数存于存储器的方法，会使传递可变数量的参数变得很困难。而且这两种方法都妨碍了可重入过程的使用。

一种更灵活的参数传送方法是使用栈。当处理器执行一次调用时，它不仅把返回地址压入栈，而且把将要传送给被调过程的参数也压入栈中。被调过程能由栈存取参数。返回时，返回参数也能放到栈中。为过程调用而存储的，包括返回地址和全部参数的栈内容称为栈帧（stack frame）。

图12-10提供了一个例子。此例有两个子过程P和Q，主程序调用过程P时在其中声明了两个局部变量x1和x2，P过程又调用Q并在Q过程中声明了局部变量y1和y2。在此图中，返回点是存于相应栈帧的第一项，接着存储的是指向先前栈帧起点的指针。如果压入栈的参数长度或数目是可变的，那么有必要保存指向先前栈帧起点的指针。



**12.5英特尔X86和ARM操作类型**

**x86操作类型**

x86提供了一系列复杂的操作类型，包括几种特殊的指令，其目的是为程序员提供一种强有力的工具，以便将高级语言程序翻译成优化的机器语言程序。大多数都是在较多机器指令集中可寻的常规指令，但有几种指令类型是为x86体系结构精心设计的，很值得考察。[CART06]的附录A列出了各种x86指令，以及每条指令的操作数和指令的执行对条件码的影响。NASM汇编语言手册的附录B，即[NASM12]提供了每条x86指令较详细的描述。这两个参考文档都在本书网站上可查到。

1. **调用/返回指令**

x86为支持过程调用/返回提供了4条指令：CALL、ENTER、LEAVE、RETURN。考察这些指令所提供的支持是有指导意义的。回想图12-10，实现过程调用/返回机制的普遍方式是使用栈帧。当调用一个新过程时，在进入新过程之前必须完成如下步骤：

·将返回点压入栈。

·将当前帧指针（frame pointer）压入栈。

·将栈指针拷贝到帧指针作为新的帧指针值。

·调整栈指针以分配帧。

CALL指令将当前指令指针值压入栈，并且通过将过程入口地址放入指令指针使跳转到过程入口。在8088和8086机器中，典型的过程开始处有如下指令序列：

PUSH EBP

  MOV EBP, ESP

SUB ESP, space-for-locals

其中EBP是帧指针，ESP是栈指针。在80286及1其后的机器中，ENTER指令以一条单独的指令完成上述全部操作。

ENTER指令被添加到指令集中，为编译器提供了直接支持。这条指令也能支持像Pascal、COBOL和Ada语言中（C和FORTRAN中未发现）的嵌套过程。不过后来发现，对于这些语言而言，有其他更好的方式来处理嵌套过程调用。另外，虽然ENTER指令（4字节）与PUSH、MOV、SUB指令序列（6字节）相比节省几个字节的内存，但实际执行时间要更长（10个时钟周期与6个时钟周期相比）。所以尽管加入ENTER指令对指令集设计者来说似乎是一个不错的想法，但它使处理器的实现变得复杂，却又几乎带不来什么好处。相比之下，我们将看到处理器设计的RISC方法会将避免使用诸如ENTER之类的复杂指令，且以更简单的指令序列来做到更有效的实现。

状态标志和条件代码状态标志是特殊寄存器中的位，可以通过某些操作设置并在条件分支指令中使用。术语条件代码指的是一个或多个状态标志的设置。在x86和许多其他体系结构中，状态标志是通过算术和比较操作设置的。大多数语言中的比较操作与减法操作一样减去两个操作数。区别在于，比较操作仅设置状态标志，而减法操作还将减法的结果存储在目标操作数中。

1. **存储管理**

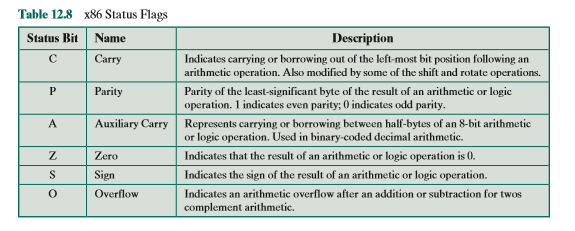
另一类指令专门用来处理存储器分段。这些是仅能由操作系统执行的特权指令。它们允许加载和读取局部或全局段表（叫做描述符表），并可检查和更改段的优先级别。

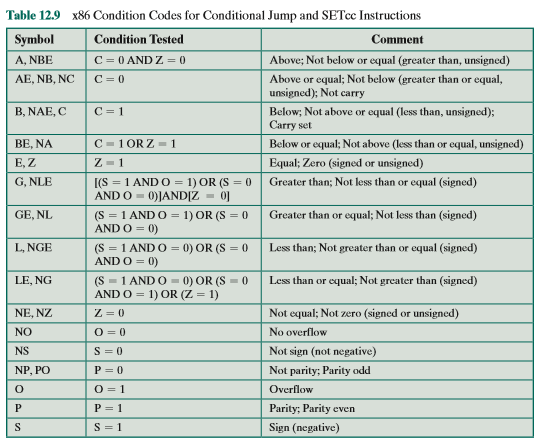
处理片内高速缓的专门指令已在第4章讨论过。

1. **状态标志和条件码**

状态标志（status flag）是可被特定操作所设置并用于条件转移指令的特殊寄存器中的位。条件码（condition code）一词指的是一个或多个状态标志的设置。在x86和很多其他体系结构中，状态标志由算术和比较操作设置。在大多数语言中，比较操作与减法操作一样，使用两个操作数相减。区别在于，比较操作只设置状态标志，而减法操作还要将结果存入目标操作数中。一些系统结构还为数据传输指令设置了状态标志。

表12-8列出了在x86上使用的状态标志。每条标志或者这些标志的组合都可以用于测试条件跳转。表12-9列出了已经定义了的条件跳转操作码的条件代码（状态标志值的组合）。





这张表有几个值得注意的地方。首先，我们可能想要测试两个操作数以确定一个是否大于另一个。但这取决于数是无符号数还是有符号数。例如，8位数11111111大于00000000是在把它们看成无符号数时才成立（255 > 0），若把它们看成是补码数则前者小于后者（-1 < 0）。于是多数汇编语言都是引入两组术语来区别这两种情况：若我们将两个数字作为有符号整数来比较，则使用小于（less than）和大于（greater than）术语；若作为无符号整数来比较，则使用低于（below）和高于（above）术语。

第二个需要关注的是比较有符号整数的复杂性。如果（1）符号位为0而且没有上溢（S = 0 AND O = 0），结果是大于或等于0；或者（2）符号位是1但有上溢（S = 1 AND O = 1）研究一下图10-4应该会使你确信，为各种带符号操作所测试的条件是合适的。

**4. x86 SMID指令**

1996年，Intel开始将MMX技术引入奔腾生产线。MMX是一组针对多媒体任务的高度优化的指令。新指令共有57条，以一种SIMD（single-instruction-multiple-data，单指令多数据）方式来处理数据。于是能一次在多个数据元素上同时完成加、乘这样的运算。一般每条指令执行只占用一个时钟周期。对于合适的应用，与不使用MMX指令[ATKI96]相比，这些快速的并行操作能产生2~8倍的加速效果。随着x86体系结构推出64位的处理器，Intel也扩展了这些指令，使它们能处理双四字（128位）操作数和浮点运算。本小节中将介绍MMX的特点。

MMX的设计重心在于多媒体程序。常规的指令是面向32位和64位数据操作的，而视频和音频数据一般是由8位或16位这样小的数据类型构成的大阵列组成。例如，在图像或视频中，每一屏都是由像素（pixel）[[2]](#footnote-2)点组成，每个像素，或者每个像素的每个颜色分量（红、绿、蓝）都由8位数据表示。声音采样一般量化成16位数据。对于某些3D图形，基本数据类型普遍为32位。为对这些长度的数据提供并行操作的便利，MMX中定义了3种新数据类型。每种数据类型都是64位长，由多个小的定点整数字段所组成。这三种新类型是；

**·压缩字节型：**8个字节打包成一个4位长的数据。

**·压缩字型**：4个字打包成一个64位长的数据。

**·压缩双字型**：2个32位的双字打包成一个4位长的数据。

表12-10列出了MMX指令集，其中大多数指令涉及字节、字、双字上的并行操作。例如，PSLLW指令完成了压缩字型每个字的逻辑左移；PADDB指令以两个压缩字节型数据为输入，对它们的每一位上并行进行字节对字节加法，输出一个压缩字节型数据。

新指令的突出特性是针对字节或16位字操作数引入了饱和算法（saturation arithmetic）。使用普通的无符号算法时，当操作溢出时（即最高有效位有向上进位），此额外位被舍掉。这被称为环绕算法（wraparound arithemetic），因为从效果上看，舍掉进位是两个数加法之和会小于被加的两个数。例如，考虑两个十六位进制数F0000h和30000h相加：

因为截断的效果可以是，例如，产生比两个输入操作数小的加法结果。考虑两个单词的加法，分别为十六进制、f000和h。

F0000h＝1111 0000 0000 0000

+3000 0h＝0011 0000 0000 0000

10010 0000 0000 0000＝2000h

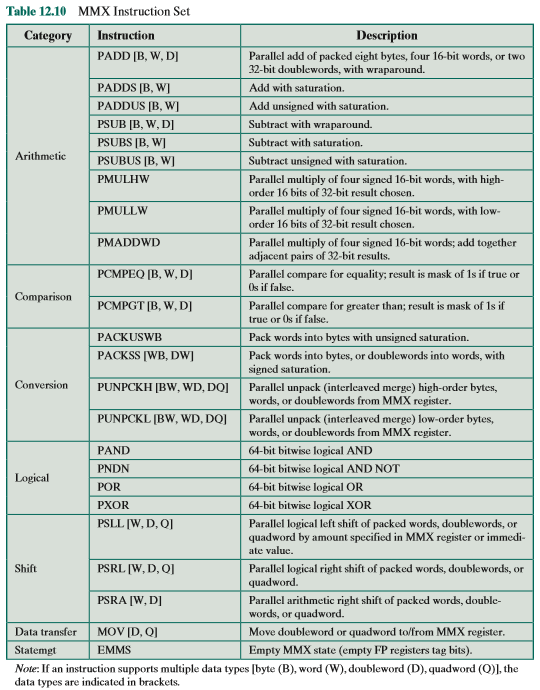
如果这两个数字代表像素的亮度（数值越大越亮），则两个暗点的组合反而更亮了，显然这不符合预期。在饱和算法中，如果加法导致了溢出，减法导致了下溢，那么结果被分别设置成可表示的最大值或最小值。对于上面这个例子，使用饱和算法则有：

F0000h＝1111 0000 0000 0000

+30000h＝0011 0000 0000 0000

10010 0000 0000 0000

1111 1111 1111 1111＝FFFFh



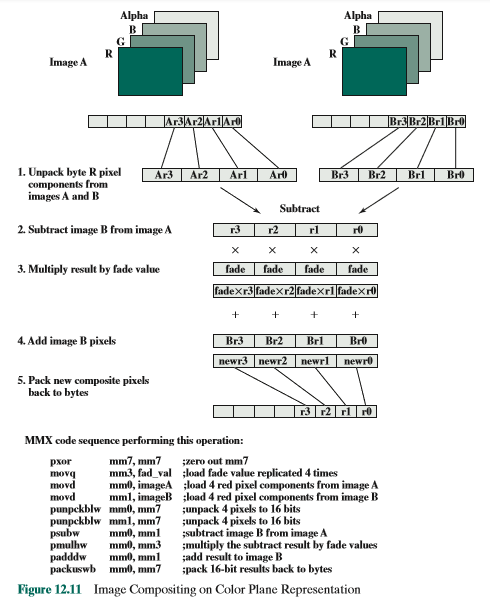
注意：如果一条指令支持多种数据类型[字节(B)、字(W)、双字(D)、四字(Q)]，则数据类型由方括号中字母指出。

为感受一下使用MMX指令的好处，让我们考察一个选自[PELE97]的例子。一般视频应用程序提供淡出、淡入效果，即一幅图像逐渐溶解成另一幅图像。两个图像以一种加权平均方式相组合：

结果像素＝A\_像素×渐变 + B\_像素 ×（ 1 – 渐变）

对A、B俩图像的每个像素位置进行上述计算。当“渐变”值由1逐渐变为0（可用相应的8位整数分成255阶），则产生一系列的图像帧，实现了由A图像淡化转成B图像的效果。

图12-11表示一组像素所需的步骤序列。8位像素分量被转换成16位元素以适应MMX的16位乘法指令。若这些图像使用640 × 480的分辨率，渐变技术使用全部的255个阶值，则使用MMX只需5.35亿条指令，而不使用MMX却要14亿条指令才能完成同样的计算[INTE98b]。



**ARM操作类型**

ARM提供了一大集合的操作类型。主要类型如下：

**·装载和保存指令**：在ARM体系结构中，只有装载（Load）的存保存（Store）指令能访问内存，算术和逻辑指令只对寄存器和指令中的立即数进行操作。这一局限是精简指令集计算机（RISC）的设计特征。在第15章我们将进一步探讨这点。ARM体系结构支持两大类装载和保存指令，一种转载和保存单个寄存器数据到内存，另一种装载和保存一对寄存器的数据到内存：（1）装载或保存一个32位字或一个8位无符号字节；（2）装载或保存一个16位无符号半字或一个8位字节。

**·分支指令**：ARM支持分支指令，允许条件分支指令向前或向后跳转最多32MB。子程序调用是通过修改标准的分支指令来实现。因为可以向前或向后跳转32MB的距离，分支链接（Branch and Link， BL）指令把本指令的后继指令地址（返回地址）保存到LR（R14）寄存器。分支的确定是通过指令中的4位条件码字段完成的。

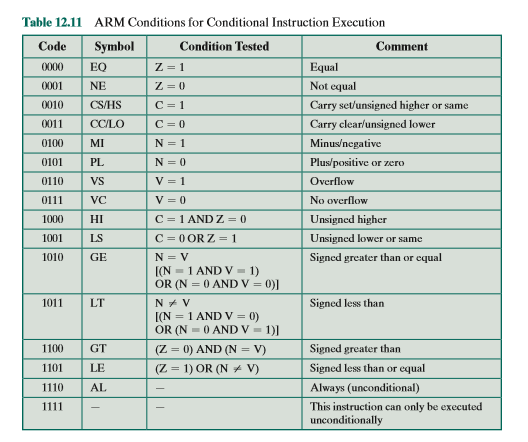
**·数据处理指令**：这一类指令包括逻辑指令（AND、OR、XOR），加法和减法指令，以及测试和比较指令。

**·乘法指令**：整数乘法指令对字或半字操作数进行计算，并产生普通或较长的结果。例如，以32位操作数作为输入，产生64位结果的乘法指令。

**·并行加法和减法指令**：除了普通的数据处理和乘法指令外，还有一组并行加法和减法指令，其中指令两个操作数的对应部分同时进行运算。例如，ADD16把两个寄存器的上半字相加，产生结果的上半字，同时把这两个寄存器的下半字相加，产生结果的下半字。这些指令类似于x86的MMX指令，对于图像处理应用很有用。

**·扩展指令**：这是用于解压缩数据的指令，它们通过符号扩展或填零扩展的方式，把字节扩展为半字或字，把半字扩展为字。

**·状态寄存器存取指令**：ARM提供了读写状态寄存器中特定位的指令。



**条件码（Condition Code）**

ARM体系结构定义了4个条件标志（Condition Flag），被保存在程序状态寄存器中： N、Z、C和V（负标志、零标志、进位标注与溢出标志），其含义与x86中的S，Z，C和V标志一样。这4个标志的值构成了ARM处理器的条件码。表12-11显示了条件执行所依赖的条件组合。

ARM中使用条件码有两个不同寻常之处：

（1）所有指令，不仅是分支指令，都有条件码字段。这意味着所有的指令都可以根据条件决定是否执行。实际上意味着除了1110和1111这两个条件标志的组合外，其他任何指令条件码字段的条件标志组合都是仅当该指令条件满足时才能执行。

（2）所有的数据处理指令（算术、逻辑）都包含过一个S位，指出该指令是否会修改条件标志。

按条件执行以及条件标志设置控制有利于设计更短的程序，从而占用更少的内存。另一方面，由于所有指令都带4位条件码字段，这意味着32位指令的操作码和操作数字段的可用位数就少了。不过ARM是一种RISC方式设计的处理器，更多地使用寄存器寻址，因此条件码的开销是可接受的。

**12.6个关键词、思考题和习题**

**关键词**

accumulator：地址

arithmetic shift：算术移位

bi-endian：双端

big endian：大端

branch：分支

conditional branch：条件分支

instruction set指令集流行

jump跳转

little endian

logical shift：逻辑移位

machine instruction：机器指令

operand：操作数

operation：操作，运算

packed decimal：压缩的十进制数

pop：出栈

procedure call：过程调用

procedure return：过程返回

push：入栈

reentrant procedure：可重入过程

rotate：循环移位，旋转

skip：跳步

stack：栈

**思考题**

12.1如何用高级语言和机器语言表达操作？

12.2列出并简要讨论四种类型的指令类型。

12.3如果一个指令包含四个地址，每条地址的目的会是什么？

12.4列出并简要说明五个重要的指令集设计问题。

12.5在机器指令集中哪些类型的操作数比较典型？

12.6 IRA字符码与压缩十进制表示法之间的关系是什么？

12.7算术移位和逻辑移位之间的区别是什么？

12.8为什么需要控制指令的转移？

12.9列出并简要说明在条件分支指令中生成要测试的条件的两种常见方法。

12.10程序嵌套（nesting of procedure）的含义是什么？

12.11列出用于存储过程返回（procedure return）的返回地址的三个可能的位置.

12.12比较运算和减法运算的区别是什么？

12.13定义像素（pixels）一词。

 12.14简要讨论了ARM中构成条件码的标志。

**习题**

12.1以十六进制数表示：

（a）压缩十进制数96。

（b）ASCII字符96。

12.2对于下列每组十进制数，给出其十进制值：

（a） 1000 1100 0110 0101

（b） 0001 0111

（c） 0010 0011 0010 0100

12.3若给定的微处理器的字长为16位。则如下表示法所能表示的最小和最大整数各是什么？

（a）无符号

（b）符号-幅值

（c）1的补码

（d）2的补码

（e）无符号压缩十进制

（f）有符号压缩十进制

12.4许多处理器提供逻辑，用于对填充的十进制数执行算术。虽然十进制运算的规则与二进制运算的规则相似，但如果使用二进制逻辑，十进制结果可能需要对单个数字进行一些校正。

考虑两个无符号数的十进制加法。如果每条数字包含N个数字，那么每条数字有4N位。这两个数字要用二进制加法器来加法。建议一个简单的改正结果的规则。以这种方式对数字1698和1786执行加法。

12.5十进制数X的十个补数被定义为10N-X，其中N是数字中的十进制数字的数目。描述使用十进制补数表示执行十进制减法。通过（21998）10减去（11998）10来说明该过程。

12.6通过编写计算程序对零、一、二、三地址机器进行比较

x=（a+b\*c）/（d -e\*f）

四台机器中的每一台。可供使用的指令如下：

12.7考虑一个假设的计算机，它的指令集只有两个n位的仪器。第一位指定操作码，其余位指定主存储器的2n-1n位字之一。这两条指令如下：

从累加器中减去位置X的内容，并将结果存储在位置X和累加器中。

在程序计数器中跳X地址X。

主存储器中的单词可以包含指令或二进制数作为两个补码符号。通过指定如何对以下操作进行编程，证明该指令库合理地完成了：

a.数据传输：位置X到累加器，累加器到位置X。

添加：将位置X的内容添加到累加器中。

C.条件分支。

逻辑或。

e. I/O操作。

12.8 Skip（考虑其两种类型中的任何一种）和Jump（考虑其三种类型中的任何一种）都是用于传输控制的指令。解释这两种指令的区别。

12.9在第12.4节中指出，当没有溢出时，算术左移位和逻辑左移位都对应于乘以2，并且如果发生溢出，则算术和逻辑左移位操作产生不同的结果，但是算术左移位保留si。GN的数量。证明这些语句对于5位的两个补整数是正确的。

“H”表示十六进制。AL是一个8位寄存器，它保存两个无符号8位整数相加的结果。AF是如果在相加的结果中有从位3到位4的进位，则设置的标志。如果有从7位到位8的进位，则CF是一个标志集。解释DAA指令执行的函数。

12.13 x86比较指令(CMP)从目的地操作数中减去源操作数；它更新状态标志(C、P、A、Z、S、O)，但不改变两个操作数。CMP指令可用于确定目标操作符是否大于、等于或小于源操作数。

假设这两个操作数被当作无符号整数。显示哪些状态标志与确定两个整数的相对大小有关，以及标志的哪些值对应于大于、等于或小于。

假设这两个操作数被当作两个补有符号整数。显示哪些状态标志与确定两个整数的相对大小有关，以及标志的哪些值对应于大于、等于或小于。

c.CMP指令后面可以跟有条件跳转(Jcc)或设置控制(SETcc)指令，其中c c引用表12.11中列出的16个条件之一。证明测试有符号数字比较的条件是正确的。

12.14假设我们希望将x86 CMP指令应用于32位操作数，这些操作数以浮点格式包含数字。为了获得正确的结果，在下列方面必须满足什么要求？

a.有效域、符号域和指数域的相对位置。

b值的表示为零。

指数的表示。

IEEE格式是否满足这些要求？解释。

12.15许多微处理器指令集包括测试条件的指令，如果条件为真，则设置目标操作数。示例包括x86上的SETcc、摩托罗拉MC68000上的Scc以及国家NS32000上的Scond。

这些指令有一些不同之处：

■SETcc和Sc只对字节进行操作，而Scond对字节、单词和双字操作数进行操作。

■SETcc和SCOND将操作数设置为整数1（如果为真）和零（如果为假）。Scc将字节设置为所有二进制字节（如果为真），如果为假，则设置为零。这些差异的相对优点和缺点是什么？

这些指令都不设置任何条件代码标志，因此需要对指令的结果进行显式测试来确定其值。讨论是否应根据此指令设置条件代码。

c.一个简单的IF语句，例如IF a 7 b THEN，可以使用数值表示方法来实现，也就是说，使布尔值显化，而不是控制流，该方法通过程序中到达的点来表示布尔表达式的值。编译器可以用以下x86代码实现7ssb THEN：

子寄存器CX，CX；设置寄存器CX至0

MOV AX，B；移动位置B到寄存器AX的内容

CMP AX，A；比较寄存器AX和位置A的内容

如果A…B跳

C.CX；添加1到寄存器CX的内容

如果Cx等于0，则跳转；

然后出来

（A 7 B）的结果是保存在寄存器中并在稍后在刚刚显示的代码流的上下文之外可用的布尔值。为此使用寄存器CX是很方便的，因为许多分支和循环操作码都有CX的内置测试。

显示使用SETcc指令的替代实现，它节省了内存和执行时间。（提示：除了SETcc之外，不需要其他新的x86指令。）

现在考虑高级语言声明：

a=（b 7 c）或（d＝f）

编译器可能生成以下代码：

移动位置B的内容以注册EAX

CMP EAX，C；比较寄存器EAX和位置C的内容

MOV BL，0；0代表假

JILE N1；如果（b…c）跳转

MOV BL，1；1代表假

1- MOV EAX，D

化学机械抛光

MOV BH，0

JNE N2

MOV BH，1

N2或BL，BH

显示使用SETcc指令的替代实现，它节省了内存和执行时间。

12.16假设两个寄存器包含以下十六进制值：AB0890C2、4598EE50。使用MMX指令添加它们的结果是：

填充字节。

包装词。

假设不使用饱和算法。

数学公式通常用中缀表示法来表示，其中二进制运算符出现在操作数之间。另一种技术称为反向Pol.（后缀）符号，其中操作符跟随它的两个操作数。详情见附录I。将下列公式从逆波兰转换为中缀：

附录I指出，如果堆栈仅由处理器用于诸如过程处理的目的，那么在指令集中就没有面向堆栈的指令。如果没有面向堆栈的指令，处理器如何能够将堆栈用于任何目的？

12.17

12.18

Ab+c+d\*

Ab/CD/+

ACDD+\*\*/

d. ABCDE+F/+G- H /\*+

12.19将下列公式从中缀转换为反向波兰：

A+B+C+D+E

b.（a+b）\*（c+d）+e

C.（a\*b）+（c\*d）+e

D（A- B）\*（（C-D\*E）/F）/G＊H

12.20使用Dijkstra算法将表达式A+B-C转换为后缀符号。显示所涉及的步骤。结果等于（A+B）-C还是A+（B-C）？这有关系吗？

12.21使用附录I中定义的中缀到后缀的转换算法，显示将图1.3的表达式转换为后缀的步骤。使用类似于图I.5的预处理。

12.22使用与图1.4类似的演示文稿，显示图1.5中表达式的计算。

12.23重新绘制图12.13中的.-endian布局，以便字节显示为数字

在大底线布局中。也就是说，以64位的行显示内存，其中从左到右、从上到下列出了字节。

12.24对于以下数据结构，使用图12.13的格式绘制大端点和小端点布局，并对结果进行评论。

当在小字节模式下运行时。当将数据结构从大端子转换为小端子时，处理器可以自由地实现真正的字节交换机制或使用某种地址修改机制。当前的Power处理器都是默认的大端计算机，使用地址修改将数据视为小端计算机。

考虑图12.13中定义的结构S。图中右下角的布局显示了处理器看到的结构s。事实上，如果结构以.-endian模式编译，那么它在内存中的布局如图12.12所示。

图12.12电力架构小

记忆中的记忆结构

解释所涉及的映射，描述实现映射的简单方法，并讨论此方法的有效性。

12.26编写一个小程序确定机器的末端并报告结果。

在可用的计算机上运行程序，并交出输出。

12.27 MIPS处理器可以设置为在大端子或小端子模式下操作。考虑一下Load Byte Unsigned(LBU)指令，它将一个字节从存储器中加载到寄存器的低阶8位中，并将寄存器的高阶24位填充为零。在MIPS参考手册中使用寄存器传输语言给出LBU的描述，如

其中byte指有效地址的两个低阶位，mem指从内存加载的值。在手册中，使用以下两个单词中的一个代替单词CONDIT.：BigEndian、LittleEndian。使用哪个词？

12.28大多数但不是全部，处理器在字节内使用大或小尾数位排序，这与多字节标量内的字节的大或小尾数排序是一致的。让我们考虑一下摩托罗拉68030，它使用大端字节排序。68030中有关格式的文档说明令人困惑。用户手册解释说，位字段的位排序与整数的位排序相反。大多数位字段操作使用一个endian排序，但是少数位字段操作需要相反的排序。以下来自用户手册的描述描述了大多数位字段操作：

位操作数由在内存中选择一个字节的基本地址（基本字节）和选择该字节中的一个比特的比特号指定。最高有效位是位七。位字段操作数由：（1）在内存中选择一个字节的基本地址；（2）位字段偏移量，该偏移量指示位字段相对于基字节的最高有效位的最左边（基本）位；（3）位字段宽度，该位字段宽度确定t的右边有多少位。基本字节位于位字段中。基本字节的最高有效位是位字段偏移0，基本字节的最低有效位是位字段偏移7。

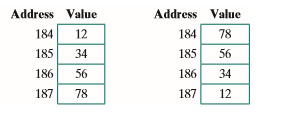
这些指令使用大尾数位排序还是小尾数位排序？

**附录12A小端，大端，和双端**

有一个恼人且奇怪的现象是字中的字节和字节中的位如何被表示及引用的问题。我们首先研究字节排序问题，然后再考虑位排序问题。

**12A.1字节排序**

Cohen首次在著作中[COHE81]讨论了端序（endianness）的概念。就字节而言，端序与多字节标量值的字节排序相关。用例子更能说明问题之所在。假定我们有32位的十六进制值12345678，并且它存储在字节位置184处的32位字字节可寻址存储器中。此值由4个字节组成，最低有效字节的值是78，最高为12。存储此值有两种方式：

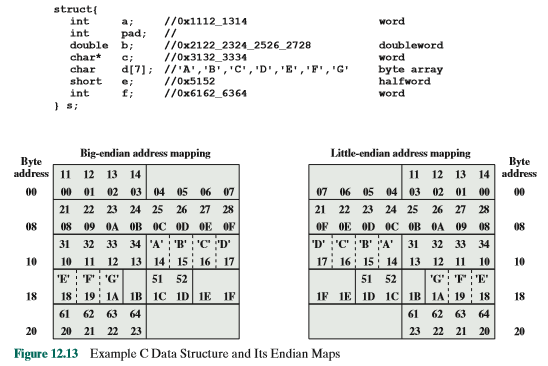


左边的映射方式是将最高有效字节存于最低数值的字节地址中，称为**大端序**（big-endian ordering），它等同于西方文化语言中的从左到右书写顺序。右边的映射方式是将最低有效字节存于最低数值的字节地址中，称为**小端序**（little-endian ordering），它使我们联想起算术单元中的从右到左的算术运算次序[[3]](#footnote-3)。对于一个给定的多字节标量值，大端和小端映射方式的字节排列彼此正好相反。

当需要将多字节实体作为具有单个地址的单个数据项来处理时，端序的概念就出现了，尽管它是由更小的可寻址单元组成。某些机器，像Intel 80x86、x86、VAX和Alpha，是小端的机器，而像IBMS370/390、Motorola的680x0、Sun SPARC和大多数RISC机器采用的是大端。当数据由一种端序类型的机器传送到另一类机器时，以及当程序试图操纵多字节标量的个别字节或个别位时，就会出现问题。

端序特性不扩展到单个数据单元之外。在任何机器中，像文件、数据结构和阵列这类的集合都是由多个数据单元组成，每个数据单元都有自己的端序。于是，将存储器数据块由一种风格的端序转换成另一种风格时，需要了解数据结构。

图12-13说明端序如何确定寻址和字节次序。图中顶部的C结构含有几种数据类型。左下部的存储器排放情况来自大端机器编译这个结构的结果，右下部是小端机器的编译结果。无论是哪种情况，存储器显示为一系列64位的行。对于大端情况，存储器安排成从左到右、从上到下；而对于小端情况，存储器安排成从右到左、从上到下。注意，这些安排是任意的。无论使用哪种映射方式，在一行内可使用从左到右也可使用从右到左的安排方式；这只是一种表述的形式，不是存储器指定的。实际上在查看某类机器的编程手册时可发现令人困惑的各种表述，甚至在同一本手册中也是这样。

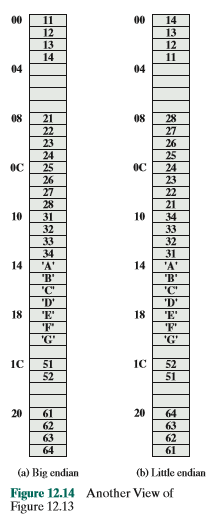


我们针对这个数据结构可以有以下几点观察：

**·**在两种策略中每个数据项有同样的地址。例如，有十六进制数2122232425262728的双字的地址是08。

**·**在任何一个给定的多字节标量值中，小端的字节排序是大端的反序，反之亦然。

**·**端序不影响结构中数据项的次序。于是，4字符的字c展示出字节的反序，但7字符的字节数组d却不是。因此，d的各个元素的地址在两种结构中是相同的。

当我们把存储器视作一个垂直的字节队列时，或许更能清除地说明端序的效果，如图12-14所示。

至于哪种端序更优并没有达成共识[[4]](#footnote-4)，下列观点更偏好大端的风格：

**·字符串排序（sorting）**：在比较整数排列的字符串时，大端机器更容易些；整数ALU能并行比较多个字节。

**·十进制/IRA打印输出：**所有值都能从左到右打印而不会引起混乱。

**·一致的次序：**大端处理器以同样的次序存储它的整数和字符串（最高有效字节最先存储）。

以下观点偏爱小端风格：

**·整数地址转换：**当一个大端处理器需要转换32位整数地址到16位整数地址时，必须执行加法运算，以便于使用最低有效字节。

**·算术：**采用小端风格完成高精度算术更容易些；你不必找到最低有效字节或反向移动。

两种风格难分高下，并且端序风格的选取经常更多考虑兼容以前的机器而不是其他因素。

PowerPC是一个既支持大端模式又支持小端模式的双序机器。这种结构允许软件开发人员在从其他机器迁移操作系统和应用时选取任一种端模式。操作系统建立处理器执行时使用的端模式。一旦模式被选定，所有后续的装载和保存都有这种模式下的存储器寻址方式所确定。为支持硬件这种特点，机器状态寄存器（Machine State Register，MSR）中的对应两位由操作系统视作进程状态的一部分来维护。以其中一位用来指定内核运行时所采用的端模式；另一位用来指定处理器当前操作的模式。于是模式可依不同进程来改变。

**12A.2 位排序**

对于字节中的位排序，迎面而来两个问题：

（1）将第1位记为0还是记为1？

（2）把此最低位号是指派给字节的最低有效位（小端）还是指派给字节的最高有效位（大端）？

这些问题在所有机器上并非都以同样的方式回答。的确，在某些机器上，不同的情况下也有不同的答案。而且字节内大端或小端位次序的选取与多字节标量中的大端或小端字节次序的选取并不总是一致的。当操作个别位时，程序员必须关注这些问题。

  另一个值得关注的领域是数据经由位串行线路传输的情况。当单个字节被传输时，系统是先传输最高有效位还是先传输最低有效位？设计者必须确保接受的位能被正确地处理。这个问题的讨论请参见[JAME90]。

1. 教科书经常将其称为二进制编码的十进制数（Binary Coded Decimal，BCD）。严格地说，BCD指的是用4位编码十进制数字，而压缩的十进制数指的是用一字节存入两个BCD编码的十进制数。 [↑](#footnote-ref-1)
2. 像素是能被指定一定灰度的数字图像的最小元素；它也等效于点阵图形表示中的像点。 [↑](#footnote-ref-2)
3. Big endian和little endian的术语源于乔纳森·斯威夫特的《格列夫游记》第1部分的第4章，它们指的是两个组织间的一场宗教争论，一方说应在小头打破鸡蛋，另一方说应在大头打破鸡蛋。 [↑](#footnote-ref-3)
4. 在《格列佛游记》一书中，争论的两个组织共同尊重的预言家说：“所有真实信徒都将在方便的一段打破他的鸡蛋。”无济于事啊！ [↑](#footnote-ref-4)