第 3 章 指令系统

指令和指令系统是计算机中最基本的概念。指令是指示计算机执行某些操作的命令，一台计算机的所有指令的集合构成该机的指令系统，也称指令集。指令系统是计算机的主要属性，位于硬件和软件的交界面上。本章将讨论一般计算机的指令系统所涉及的基本问题。

3.1 指令格式

一台计算机指令格式的选择和确定涉及多方面的因素，如指令长度、地址码结构以及操作码结构等，是一个很复杂的问题，它与计算机系统结构、数据表示方法、指令功能设计等都密切相关。

3.1.1 机器指令的基本格式

一条指令就是机器语言的一个语句，它是一组有意义的二进制代码，指令的基本格式如下：

操作码字段	地址码字段

其中，操作码指明了指令的操作性质及功能，地址码则给出了操作数的地址。

指令的长度是指一条指令中所包含的二进制代码的位数，它取决于操作码字段的长度、操作数地址的个数及长度。指令长度与机器字长没有固定的关系，它可以等于机器字长，也可以大于或小于机器字长。在字长较短的小型、微型计算机中，大多数指令的长度可能大于机器的字长；而在字长较长的大型、中型计算机中，大多数指令的长度则往往小于或等于机器的字长。通常，把指令长度等于机器字长的指令称为单字长指令；指令长度等于半个机器字长的指令称为半字长指令；指令长度等于两个机器字长的指令称为双字长指令。

在一个指令系统中，若所有指令的长度都是相等的，就称该指令系统为定长指令字结构。定长结构指令系统控制简单，但不够灵活。若各种指令的长度随指令功能而异，就称该指令系统为变长指令字结构。现代计算机广泛采用变长指令字结构，指令的长度能短则短，需长则长，如 80x86 的指令长度从一个字节到十几个字节不等。变长结构指令系统灵活，能充分利用指令长度，但指令的控制较为复杂。

3.1.2 地址码结构

计算机执行一条指令所需要的全部信息都必须包含在指令中。对于一般的双操作数运算类指令来说,除去操作码(Operation Code)之外,指令还应包含以下信息:

① 第一操作数地址,用 A_1 表示。

② 第二操作数地址,用 A_2 表示。

③ 操作结果存放地址,用 A_3 表示。

④ 下一条将要执行指令的地址,用 A_4 表示。

这些信息可以在指令中明显地给出,称为显地址;也可以依照某种事先的约定,用隐含的方式给出,称为隐地址。下面从地址结构的角度来介绍几种指令格式。

1. 四地址指令

前述的4个地址信息都在地址字段中明显地给出,其指令的格式为:

OP	A_1	A_2	A_3	A_4

指令的含义:

$$(A_1)\,OP\,(A_2)\rightarrow A_3$$

$$A_4 = \text{下一条将要执行指令的地址}$$

其中,OP 表示具体的操作,A_i 表示地址,(A_i) 表示存放于该地址中的内容。

这种格式的主要优点是直观,下一条指令的地址明显。但是,最严重的缺点是指令的长度太长,如果每个地址为16位,整个地址码字段就要长达64位,所以这种格式是不切实际的。

2. 三地址指令

正常情况下,大多数指令按顺序依次被从主存中取出来执行,只有在遇到转移指令时,程序的执行顺序才会改变。因此,可以用一个程序计数器(Program Counter,PC)来存放指令地址。通常每执行一条指令,PC 就自动加1(设每条指令只占一个主存单元),直接得到将要执行的下一条指令的地址。这样,指令中就不必再明显地给出下一条指令的地址了。三地址指令格式为:

OP	A_1	A_2	A_3

指令的含义:

$$(A_1)\,OP\,(A_2)\rightarrow A_3$$

$$(PC)+1\rightarrow PC(\text{隐含})$$

执行一条三地址的双操作数运算指令,至少需要访问4次主存。第一次取指令本身,第二次取第一操作数,第三次取第二操作数,第四次保存运算结果。

这种格式省去了一个地址,但指令长度仍比较长,所以只在字长较长的大型、中型计算机中使用,小型、微型计算机中很少使用。

3. 二地址指令

三地址指令执行完后，主存中的两个操作数均不会被破坏。然而，通常并不一定需要完整的保留两个操作数。例如，可让第一操作数地址同时兼作存放结果的地址（目的地址），这样即得到了二地址指令，其格式为：

OP	A_1	A_2

指令的含义：

$$(A_1)\,OP\,(A_2)\rightarrow A_1$$

$$(PC)+1\rightarrow PC(\text{隐含})$$

其中，A_1为目的操作数地址，A_2为源操作数地址。

注意：指令执行之后，目的操作数地址中原存的内容已被破坏了。

执行一条二地址的双操作数运算指令，同样至少需要访问 4 次主存。

4. 一地址指令

一地址指令顾名思义只有一个显地址，它的指令格式为：

OP	A_1

一地址指令只有一个地址，那么另一个操作数来自何方呢？指令中虽未明显给出，但按事先约定，这个隐含的操作数就放在一个专门的寄存器中。因为这个寄存器在连续运算时，保存着多条指令连续操作的累计结果，故称为累加寄存器（Accumulator，Acc）。

指令的含义：

$$(Acc)\,OP\,(A_1)\rightarrow Acc$$

$$(PC)+1\rightarrow PC(\text{隐含})$$

执行一条一地址的双操作数运算指令，只需要访问两次主存。第一次取指令本身，第二次取第二操作数。第一操作数和运算结果都放在累加寄存器中，所以读取和存入都不需要访问主存。

5. 零地址指令

零地址指令格式中只有操作码字段，没有地址码字段，其格式为：

OP

零地址的算术逻辑类指令是用在堆栈计算机中的，堆栈计算机没有一般计算机中必备的通用寄存器，因此堆栈就成为提供操作数和保存运算结果的唯一场所。通常，参加算术逻辑运算的两个操作数隐含地从堆栈顶部弹出，送到运算器中进行运算，运算的结果再隐含地压入堆栈。有关堆栈的概念将在稍后介绍。

指令中地址个数的选取要考虑诸多的因素。从缩短程序长度、用户使用方便、增加操作并行度等方面来看，选用三地址指令格式较好；从缩短指令长度，减少访存次数、简化硬件设

计等方面来看,一地址指令格式较好。对于同一个问题,用三地址指令编写的程序最短,但指令长度(程序存储量)最长;而用二、一、零地址指令来编写程序,程序的长度一个比一个长,但指令的长度一个比一个短。表3-1给出了不同地址数指令的特点及适用场合。

表3-1 不同地址数指令的特点及适用场合

地址数量	程序长度	程序存储量	执行速度	适用场合
三地址	短	最大	一般	以向量、矩阵运算为主
二地址	一般	很大	很低	一般不宜采用
一地址	较长	较大	较快	连续运算,硬件结构简单
零地址	最长	最小	最低	嵌套、递归问题

前面介绍的操作数地址都是指主存单元的地址,实际上许多操作数可能是存放在通用寄存器里的。计算机在CPU中设置了相当数量的通用寄存器,用它们来暂存运算数据或中间结果,这样可以大大减少访存次数,提高计算机的处理速度。实际使用的二地址指令多为二地址R(通用寄存器)型,一般通用寄存器数量有8～32个,其地址(或称寄存器编号)有3～5位就可以了。由于二地址R型指令的地址码字段很短,且操作数就在寄存器中,所以这类指令的程序存储量最小,程序执行速度最快,在小型、微型计算机中被大量使用。

3.1.3 指令的操作码

指令系统中的每一条指令都有一个唯一确定的操作码,指令不同,其操作码的编码也不同。通常,希望用尽可能短的操作码字段来表达全部的指令。指令操作码的编码可以分为规整型和非规整型两类编码。

1. 规整型编码(定长编码)

这是一种最简单的编码方法,操作码字段的位数和位置是固定的。为了能表示整个指令系统中的全部指令,指令的操作码字段应当具有足够的位数。

假定指令系统共有 m 条指令,指令中操作码字段的位数为 N 位,则有如下关系式:

$$m \leqslant 2^N$$

所以,

$$N \geqslant \log_2 m$$

定长编码对于简化硬件设计、减少指令译码的时间是非常有利的,在字长较长的大型、中型计算机及超级小型计算机上广泛采用。例如,IBM 370机(字长32位)中采用的就是这种方式。IBM 370机的指令可分为3种不同的长度形式:半字长指令(16位)、单字长指令(32位)和一个半字长指令(48位),共有5种格式,如图3-1所示。

从图3-1可以看出,在IBM 370机中不论指令的长度为多少位,其操作码字段一律都是8位。8位操作码允许容纳256条指令。而实际上在IBM 370机中仅有183条指令,存在着极大的信息冗余,这种信息冗余的编码也称为非法操作码。

2. 非规整型编码(变长编码)

变长编码的操作码字段的位数不固定,且分散地放在指令字的不同位置上。这种方式能

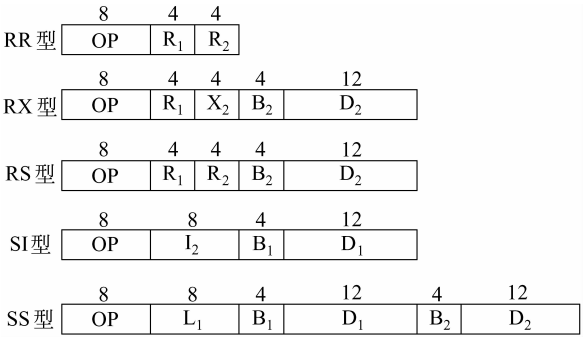


图 3-1　IBM 370 机的指令格式

够有效地压缩指令中操作码字段的平均长度，在字长较短的小型、微型计算机上广泛采用。例如，PDP-11 机(字长 16 位)中采用的就是这种方式。PDP-11 机的指令分为单字长、二字长、三字长 3 种，操作码字段占 4～16 位不等，可遍及整个指令长度，其指令格式如图 3-2 所示。

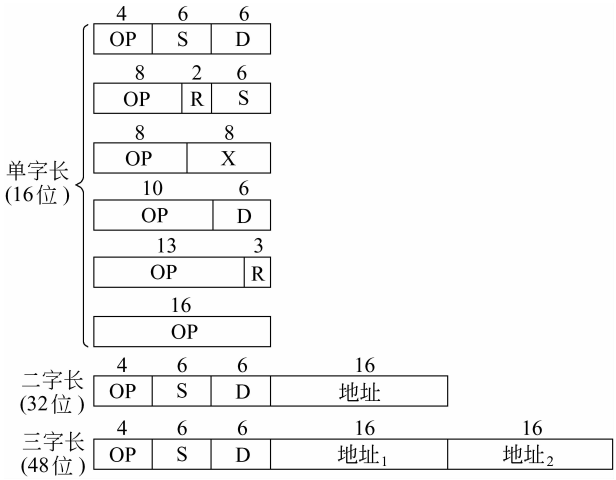
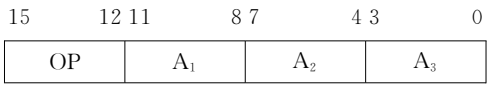


图 3-2　PDP-11 机的指令格式

显然，操作码字段的位数和位置不固定将增加指令译码和分析的难度，使得控制器的设计复杂化。

最常用的非规整型编码方式是扩展操作码法。因为如果指令长度一定，则地址码与操作码字段的长度是相互制约的。为了解决这一矛盾，让操作数地址个数多的指令(三地址指令)的操作码字段短些，操作数地址个数少的指令(一或零地址指令)的操作码字段长些，这样既能充分地利用指令的各个字段，又能在不增加指令长度的情况下扩展操作码的位数，使它能表示更多的指令。例如，设某计算机的指令长度为 16 位，操作码字段为 4 位，有 3 个 4 位的地址码字段，其格式为：



如果按照定长编码的方法，4 位操作码最多只能表示 16 条不同的三地址指令。假设指

令系统中不仅有三地址指令,还有二地址指令、一地址指令和零地址指令,利用扩展操作码法可以使在指令长度不变的情况下,指令的总数远远大于16条。例如,指令系统中要求有15条三地址指令、15条二地址指令、15条一地址指令和16条零地址指令,共61条指令。显然,只有4位操作码是不够的,解决的方法就是向地址码字段扩展操作码的位数。扩展的方法如下:

① 4位操作码的编码0000～1110定义了15条三地址指令,留下1111作为扩展窗口,与下一个4位(A_1)组成一个8位的操作码字段。

② 8位操作码的编码11110000～11111110定义了15条二地址指令,留下11111111作为扩展窗口,与下一个4位(A_2)组成一个12位的操作码字段。

③ 12位操作码的编码111111110000～111111111110定义了15条一地址指令,扩展窗口为111111111111,与A_3组成16位的操作码字段。

④ 最后,16条零地址指令由16位操作码的编码1111111111110000～1111111111111111给出。

根据指令系统的要求,扩展操作码的组合方案可以有很多种,但有以下两点要注意:

- 不允许短码是长码的前缀,即短码不能与长码的开始部分的代码相同,否则将无法保证解码的唯一性和实时性。
- 各条指令的操作码一定不能重复,而且各类指令的格式安排应统一规整。

3.2 寻址技术

所谓寻址,指的是寻找操作数的地址或下一条将要执行的指令地址,寻址技术是计算机设计中硬件对软件最早提供支持的技术之一。寻址技术包括编址方式和寻址方式。

3.2.1 编址方式

在计算机中,编址方式是指对各种存储设备进行编码的方式。

1. 编址

通常,指令中的地址码字段将指出操作数的来源和去向,而操作数则存放在相应的存储设备中。在计算机中需要编址的设备主要有CPU中的通用寄存器、主存储器和输入输出设备3种。

要对寄存器、主存储器和输入输出设备进行访问,首先必须对它们进行编址。就像一个大楼有许多房间,首先必须给每一个房间编上一个唯一的号码,人们才能据此找到需要的房间一样。

如果存储设备是CPU中的通用寄存器,那么在指令字中应给出寄存器编号;如果是主存的一个存储单元,那么在指令字中应给出该主存单元的地址;如果是输入输出设备(接口)中的一个寄存器,那么指令字中应给出设备编号或设备端口地址或设备映像地址(与主存地址统一编址时)。

2. 编址单位

目前常用的编址单位有字编址、字节编址和位编址。

(1) 字编址

字编址是实现起来最容易的一种编址方式,这是因为每个编址单位与访问单位相一致,即每个编址单位所包含的信息量(二进制位数)与访问一次寄存器、主存所获得的信息量相同。早期的大多数机器都采用这种编址方式。

在采用字编址的机器中,每执行一条指令,程序计数器加 1;每从主存中读出一个数据,地址计数器加 1。这种控制方式实现起来简单,地址信息没有任何浪费。但它的主要缺点是不支持非数值应用,而目前在计算机的实际应用领域中,非数值应用已超过数值应用。

(2) 字节编址

目前使用最普遍的编址方式是字节编址,这是为了适应非数值应用的需要。字节编址方式使编址单位与信息的基本单位(一个字节)相一致,这是它的最大优点。然而,如果主存的访问单位也是一个字节的话,那么主存的带宽就太窄了,所以编址单位和主存的访问单位是不相同的。通常主存的访问单位是编址单位的若干倍。

在采用字节编址的机器中,如果指令长度是 32 位,那么每执行完一条指令,程序计数器要加 4。如果数据字长是 32 位,当连续访问存储器时,每读写完一个数据字,地址寄存器要加 4。由此可见,字节编址方式存在着地址信息的浪费。

(3) 位编址

有部分计算机系统采用位编址方式,如 STAR-100 巨型计算机等。这种编址方式的地址信息浪费更大。

3. 指令中地址码的位数

指令格式中每个地址码的位数是与主存容量和最小寻址单位(即编址单位)有关联的。主存容量越大,所需的地址码位数就越长。对于相同容量来说,如果以字节为最小寻址单位,那么地址码的位数就需要长些,但是可以方便地对每一个字符进行处理;如果以字为最小寻址单位(假定字长为 16 位或更长),那么地址码的位数可以减少,但对字符操作比较困难。例如,设某计算机主存容量为 2^{20} 个字节,机器字长 32 位,若最小寻址单位为字节(按字节编址),其地址码应为 20 位;若最小寻址单位为字(按字编址),其地址码只需 18 位。从减少指令长度的角度看,最小寻址单位越大越好;而从对字符或位的操作是否方便的角度看,最小寻址单位越小越好。

3.2.2 指令寻址和数据寻址

寻址可以分为指令寻址和数据寻址。寻找下一条将要执行的指令地址称为指令寻址,寻找操作数的地址称为数据寻址。指令寻址比较简单,它又可以细分为顺序寻址和跳跃寻址。而数据寻址方式种类较多,其最终目的都是寻找所需要的操作数。

顺序寻址可通过程序计数器加 1,自动形成下一条指令的地址;跳跃寻址则需要通过程序转移类指令实现。

跳跃寻址的转移地址形成方式有 3 种:直接(绝对)、相对和间接寻址,它与下面介绍的数据寻址方式中的直接、相对和间接寻址是相同的,只不过寻找到的不是操作数的有效地址而是转移的有效地址。

3.2.3 基本的数据寻址方式

数据寻址方式是根据指令中给出的地址码字段寻找真实操作数地址的方式。一般情况下,由于指令长度的限制,指令中的地址码不会很长,而主存的容量却可能越来越大。以IBM PC/XT机为例,主存容量可达1MB,而指令中的地址码字段最长仅16位,仅能直接访问主存的一小部分,而无法访问整个主存空间。就是在字长很长的大型机中,即使指令中能够拿出足够的位数来作为访问整个主存空间的地址,为了灵活方便地编制程序,也需要对地址进行必要的变换。指令中地址码字段给出的地址称为形式地址(用字母A表示),这个地址有可能不能直接用来访问主存。形式地址经过某种运算而得到的能够直接访问主存的地址称为有效地址(用字母EA表示)。从形式地址生成有效地址的各种方式称为寻址方式,即:

$$\text{指令中的形式地址} \xrightarrow{\text{寻址方式}} \text{有效地址}$$

每种计算机的指令系统都有自己的一套数据寻址方式,不同计算机的寻址方式的名称和含义并不统一,下面介绍大多数计算机常用的几种基本寻址方式。

1. 立即寻址

立即寻址是一种特殊的寻址方式,指令中在操作码字段后面的部分不是通常意义上的操作数地址,而是操作数本身。也就是说,数据就包含在指令中,只要取出指令,也就取出了可以立即使用的操作数,这样的数称为立即数,其指令格式为:

这种方式的特点是:在取指令时,操作码和操作数被同时取出,不必再次访问主存,从而提高了指令的执行速度。但是,因为操作数是指令的一部分,不能被修改,而且立即数的大小受到指令长度的限制,所以这种寻址方式灵活性最差,通常用于给某一寄存器或主存单元赋初值或提供一个常数。

2. 寄存器寻址

寄存器寻址指令的地址码部分给出某一个通用寄存器的编号R_i,这个指定的寄存器中存放着操作数。寄存器寻址过程如图3-3所示,图中的IR表示指令寄存器,它的内容是从主存中取出的指令。操作数S与寄存器R_i的关系为:

$$S=(R_i)$$

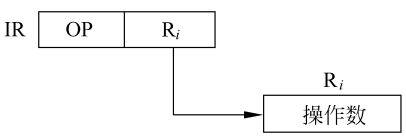


图3-3 寄存器寻址过程

这种寻址方式具有两个明显的优点:

① 从寄存器中存取数据比从主存中存取快得多。

② 由于寄存器的数量较少,其地址码字段比主存单元地址字段短得多。

这种方式可以缩短指令长度,提高指令的执行速度,几乎所有的计算机都使用了寄存器寻址方式。

3. 直接寻址

指令中地址码字段给出的地址 A 就是操作数的有效地址，即形式地址等于有效地址：EA＝A。由于这样给出的操作数地址是不能修改的，与程序本身所在的位置无关，所以又称为绝对寻址方式。图 3-4 所示为直接寻址的示意图。操作数 S 与地址码 A 的关系为：

$$S=(A)$$

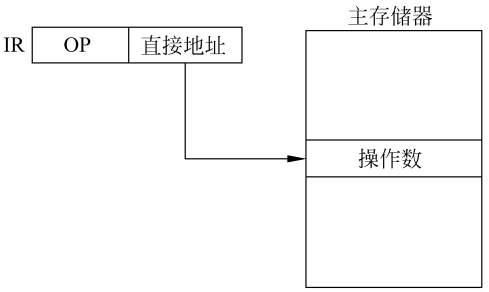


图 3-4　直接寻址过程

这种寻址方式不需做任何寻址运算，简单直观，也便于硬件实现，但地址空间受到指令中地址码字段位数的限制。

4. 间接寻址

间接寻址意味着指令中给出的地址 A 不是操作数的地址，而是存放操作数地址的主存单元的地址，简称操作数地址的地址。通常在指令格式中划出一位作为直接或间接寻址的标志位，间接寻址时标志位@＝1。

间接寻址中又有一级间接寻址和多级间接寻址之分。在一级间接寻址中，首先按指令的地址码字段先从主存中取出操作数的有效地址，即 EA＝(A)，然后再按此有效地址从主存中读出操作数，如图 3-5(a)所示。操作数 S 与地址码 A 的关系为：

$$S=((A))$$

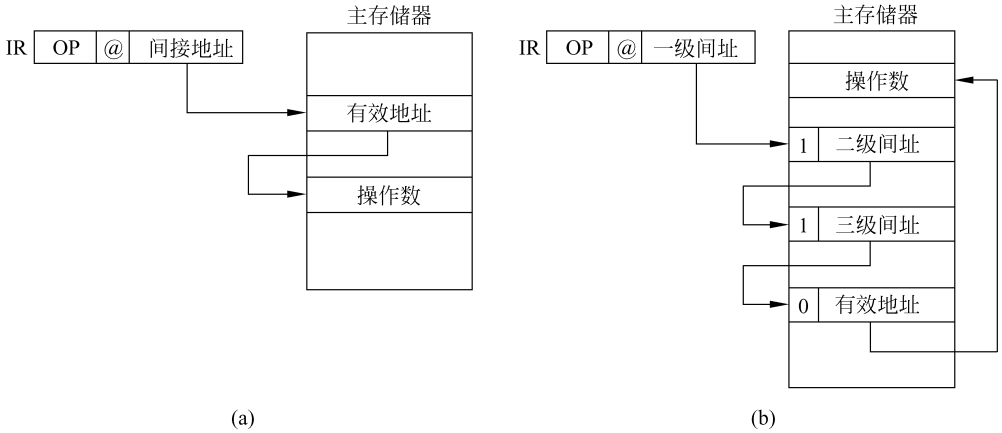


图 3-5　间接寻址过程

多级间接寻址为取得操作数需要多次访问主存,即使在找到操作数有效地址后,还需再访问一次主存才可得到真正的操作数,如图 3-5(b)所示。对于多级间接寻址来说,在寻址过程中所访问到的每个主存单元的内容中都应设有一个间址标志位。通常将这个标志放在主存单元的最高位。当该位为 1 时,表示这一主存单元中仍然是间接地址,需要继续间接寻址;当该位为 0 时,表示已经找到了有效地址,根据这个地址可以读出真正的操作数。

间接寻址要比直接寻址灵活得多,它的主要优点如下:

① 扩大了寻址范围,可用指令中的短地址访问大的主存空间。

② 可将主存单元作为程序的地址指针,用以指示操作数在主存中的位置。当操作数的地址需要改变时,不必修改指令,只须修改存放有效地址的那个主存单元的内容即可。

但是,间接寻址在取指之后至少需要两次访问主存才能取出操作数,降低了取操作数的速度。尤其是在多级间接寻址时,寻找操作数要花费相当多的时间,甚至可能发生间址循环。

5. 寄存器间接寻址

为了克服间接寻址中访存次数多的缺点,可采用寄存器间接寻址,即指令中的地址码给出某一通用寄存器的编号,在被指定的寄存器中存放操作数的有效地址,而操作数则存放在主存单元中,其寻址过程如图 3-6 所示。操作数 S 与寄存器号 R_i 的关系为:

$$S=((R_i))$$

这种寻址方式的指令较短,并且在取指后只须一次访存便可得到操作数,因此指令执行速度较间接寻址方式快,是一种使用广泛的寻址方式。

6. 变址寻址

变址寻址就是把变址寄存器 R_x 的内容与指令中给出的形式地址 A 相加,形成操作数有效地址,即 $EA=(R_x)+A$。R_x 的内容称为变址值,其寻址过程如图 3-7 所示。操作数 S 与地址码和变址寄存器的关系为:

$$S=((R_x)+A)$$

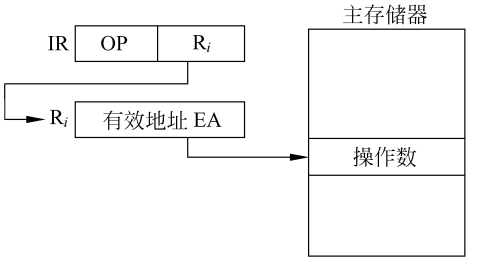


图 3-6　寄存器间接寻址过程

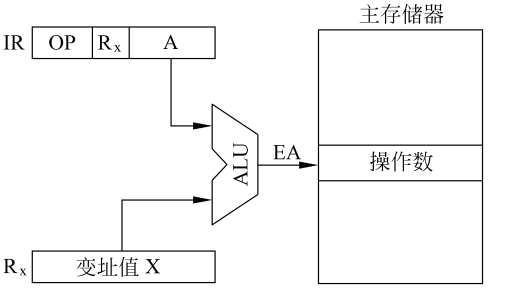


图 3-7　变址寻址过程

变址寻址是一种广泛采用的寻址方式,最典型的用法是将指令中的形式地址作为基准地址,而变址寄存器的内容作为修改量。在遇到需要频繁修改地址时,无须修改指令,只要修改变址值就可以了,这对于数组运算、字符串操作等成批数据处理是很有用的。例如,要

把一组连续存放在主存单元中的数据(首地址是 A)依次传送到另一存储区(首地址为 B)中,则只须在指令中指明两个存储区的首地址 A 和 B(形式地址),用同一变址寄存器提供修改量 K,即可实现(A+K)→B+K。变址寄存器的内容在每次传送之后自动地修改。

在具有变址寻址的指令中,除去操作码和形式地址外,还应具有变址寻址标志,当有多个变址寄存器时,还必须指明具体寻找哪一个变址寄存器。

7. 基址寻址

基址寻址是将基址寄存器 R_b 的内容与指令中给出的位移量 D 相加,形成操作数有效地址,即 $EA=(R_b)+D$。基址寄存器的内容称为基址值。指令的地址码字段是一个位移量,位移量可正、可负,如图 3-8 所示。操作数 S 与基址寄存器和地址码的关系为:

$$S=((R_b)+D)$$

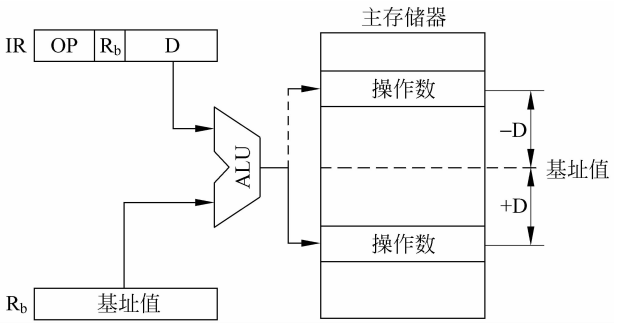


图 3-8　基址寻址过程

基址寻址原是大型计算机采用的一种技术,用来将用户的逻辑地址(用户编程时使用的地址)转化成主存的物理地址(程序在主存中的实际地址)。

基址寻址和变址寻址在形成有效地址时所用的算法是相同的,而且在一些计算机中,这两种寻址方式都是由同样的硬件来实现的。但是,它们两者实际上是有区别的。一般来说,变址寻址中变址寄存器提供修改量(可变的),而指令中提供基准值(固定的);基址寻址中基址寄存器提供基准值(固定的),而指令中提供位移量(可变的)。这两种寻址方式应用的场合也不同,变址寻址是面向用户的,用于访问字符串、向量和数组等成批数据;而基址寻址面向系统,主要用于逻辑地址和物理地址的变换,用以解决程序在主存中的再定位和扩大寻址空间等问题。在某些大型机中,基址寄存器只能由特权指令来管理,用户指令无权操作和修改。在某些小、微型计算机中,基址寻址和变址寻址实际上是合二为一的。

8. 相对寻址

相对寻址是基址寻址的一种变通,由程序计数器(PC)提供基准地址,指令中的地址码字段作为位移量 D,两者相加后得到操作数的有效地址,即 EA=(PC)+D。位移量指出的是操作数和现行指令之间的相对位置,如图 3-9 所示。

这种寻址方式有如下两个特点:

① 操作数的地址不是固定的,它随着 PC 值的变化而变化,并且与指令地址之间总是相差一个固定值。当指令地址变换时,由于其位移量不变,使得操作数与指令在可用的存储区

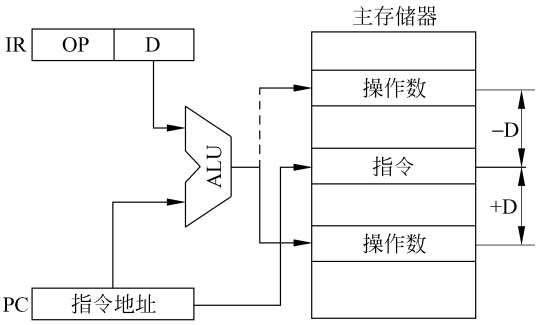


图 3-9　相对寻址过程

内一起移动,所以仍能保证程序的正确执行。采用 PC 相对寻址方式编写的程序可在主存中任意浮动,它放在主存的任何地方所执行的效果都是一样的。

② 对于指令地址而言,操作数地址可能在指令地址之前或之后,因此指令中给出的位移量可负、可正,通常用补码表示。如果位移量为 n 位,则相对寻址的寻址范围为:

$$(PC)-2^{(n-1)}\sim(PC)+2^{(n-1)}-1$$

注意:有些计算机是以当前指令地址为基准的,有些计算机是以下一条指令地址为基准的。这是因为有的机器是在当前指令执行完时,才将 PC 的内容加 1(或加增量);而有的机器是在取出当前指令后立即将 PC 的内容加 1(或加增量),使之变成下一条指令的地址。后一种方法将使位移量的计算变得比较复杂,特别是对于变字长指令更加麻烦。不过在实际应用时,位移量是由汇编程序自动形成的,程序员并不需要特别关注。

9. 页面寻址

页面寻址相当于将整个主存空间分成若干个大小相同的区,每个区称为一页,每页有若干个主存单元。例如,1 个 64KB 的存储器被划分为 256 个页面,每个页面中有 256 个字节,如图 3-10(a)所示。每页都有自己的编号,称为页面地址;页面内的每个主存单元也有自己的编号,称为页内地址。这样,存储器的有效地址就被分为两部分:前部为页面地址(在此例中占 8 位),后部为页内地址(也占 8 位)。页内地址由指令的地址码部分自动直接提供,它与页面地址通过简单的拼装连接就可得到有效地址,无须进行计算,因此寻址迅速。根据页面地址的来源不同,页面寻址又可以分成以下 3 种不同的方式。

① 基页寻址。基页地址又称零页寻址。由于页面地址全等于 0,所以有效地址 EA=0//A(//在这里表示简单拼接),操作数 S 在零页面中,如图 3-10(b)所示。基页寻址实际上就是直接寻址。

② 当前页寻址。页面地址就等于程序计数器(PC)的高位部分的内容,所以有效地址 $EA=(PC)_H//A$,操作数 S 与指令本身处于同一页面中,如图 3-10(c)所示。

③ 页寄存器寻址。页面地址取自页寄存器,与形式地址相拼接形成有效地址,如图 3-10(d)所示。

前两种方式因不需要页寄存器,所以用得较多些。有些计算机在指令格式中设置了一个页面标志位(Z/C)。Z/C=0,表示 0 页寻址;Z/C=1,表示当前页寻址。

怎样才能知道一条指令所采用的是什么寻址方式呢?为了能区分出各种不同的寻址方

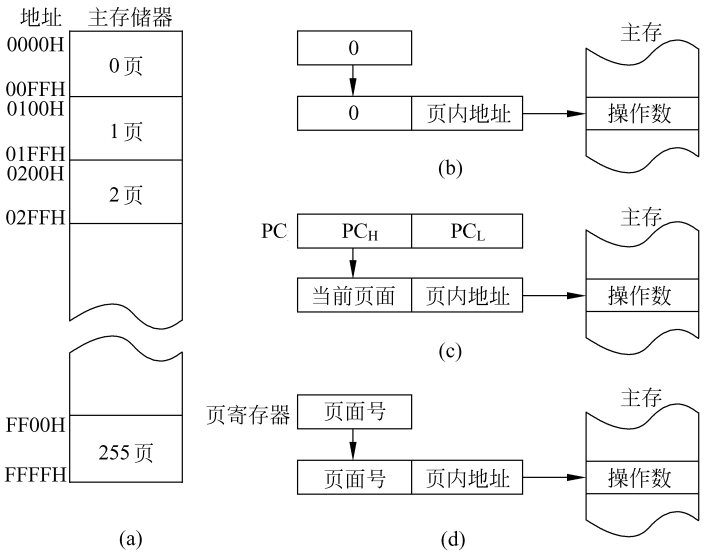


图 3-10　页面寻址

式，必须在指令中给出标识。标识的方式通常有两种：显式和隐式。显式的方法就是在指令中设置专门的寻址方式字段，用二进制编码来表明寻址方式类型，如图 3-11(a)所示；隐式的方式是由指令的操作码字段说明指令格式并隐含约定寻址方式，如图 3-11(b)所示。

注意：一条指令若有两个或两个以上的地址码时，各地址码可采用不同的寻址方式。例如，源地址采用一种寻址方式，而目的地址采用另一种寻址方式。

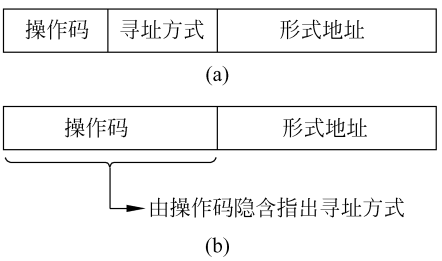


图 3-11　指令中寻址方式的表示

3.2.4　变型或组合寻址方式

前面介绍了 9 种常用的基本寻址方式，其他的寻址方式则是这 9 种寻址方式的变型或组合。

1. 自增型寄存器间址和自减型寄存器间址

这两种寻址方式实际上都是寄存器间接寻址方式的变型，通用寄存器在这里作为自动变址寄存器。

(1) 自增寻址

在自增寻址时，寄存器 R_i 的内容是有效地址，按照这个有效地址从主存中取数以后，寄存器的内容自动增量修改。在字节编址的计算机中，若指向下一个字节，寄存器的内容加 1；若指向下一个字(假设字长 16 位)，寄存器的内容加 2，如图 3-12(a)所示。

寻址操作的含义为：$EA=(R_i)$，$R_i \leftarrow (R_i)+d$。其中，EA 为有效地址，d 为修改量，通常记作$(R_i)+$，加号在括号之后，形象地表示先操作后修改。

(2) 自减寻址

自减寻址是先对寄存器 R_i 的内容自动减量修改(减 1 或减 2),修改之后的内容才是操作数的有效地址,据此可到主存中取出操作数。图 3-12(b)给出了自减寻址过程示意图。

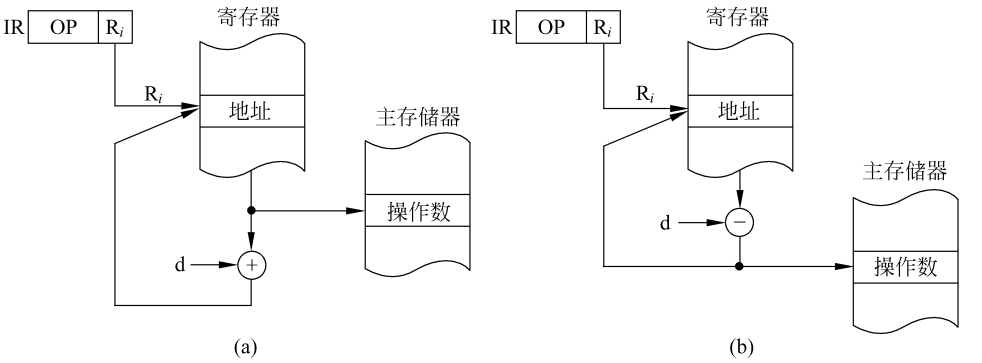


图 3-12 自增/自减寻址方式

寻址操作的含义为:$R_i \leftarrow (R_i)-d$,$EA=(R_i)$,通常记作 $-(R_i)$,减号在括号之前,形象地表示先修改后操作。自减寻址和自增寻址一起,可以使任何一个寄存器作为堆栈指针。

采用自增/减寻址最灵活的当属 MC68000 机,它具有字节、字、双字的自动增/减寻址方式。

2. 扩展变址方式

把变址和间址两种寻址方式结合起来就成为扩展变址方式,按寻址方式操作的先后顺序,有前变址和后变址两种寻址方式。

(1) 先变址后间址(前变址寻址方式)

先进行变址运算,其运算结果作为间接地址,间接地址指出的单元的内容才是有效地址。所以,有效地址 $EA=((R_x)+A)$,操作数 $S=(((R_x)+A))$。其寻址过程如图 3-13(a)所示。

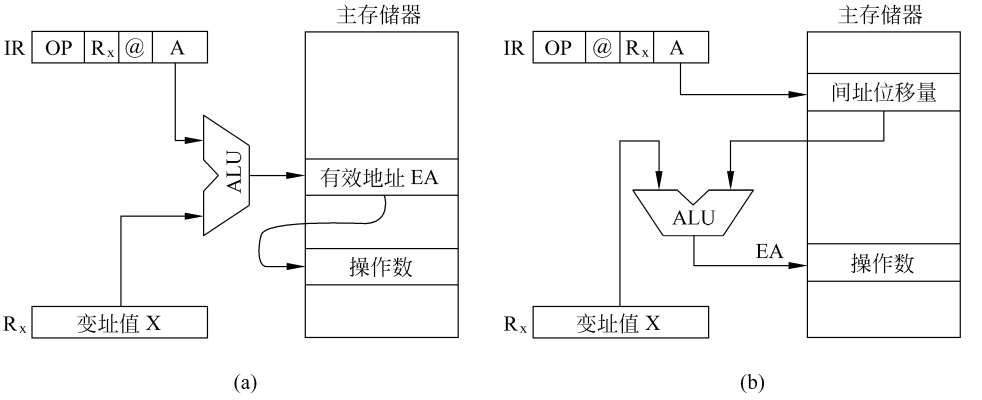


图 3-13 前/后变址寻址方式

(2) 先间址后变址(后变址寻址方式)

将指令中的地址码先进行一次间接寻址,然后再与变址值进行运算,从而得到一个有效地址。所以,有效地址 EA=(R_x)+(A),操作数 S=((R_x)+(A))。其寻址过程如图 3-13(b)所示。

3. 基址变址寻址

基址变址寻址是最灵活的一种寻址方式,此时有效地址是由基址寄存器中的值、变址寄存器中的值和位移量三者相加求得的。在这 3 项中,除位移量在指令一旦确定后就不能再修改以外,基址和变址寄存器中的内容都可以改变。

$$EA=(R_b)+(R_x)+D$$

其中,R_b 为基址寄存器,R_x 为变址寄存器,D 为位移量。

IBM 370 机中就有这种寻址方式。实际上,R_b 和 R_x 并不单独存在,通常借用 16 个通用寄存器中的 15 个(0 寄存器除外)来作为 R_b 或 R_x。上式 3 项中的任何一项都可以缺省。

基址变址寻址方式在 Intel 80x86 中是最基本的寻址方式,其他多种方式可由它派生出来。基址寄存器(BX 或 BP)、变址寄存器(SI 或 DI)及位移量都可以缺省,位移量允许是 8 位或 16 位的带符号数。

$$EA=\begin{Bmatrix}(BX)\\(BP)\end{Bmatrix}+\begin{Bmatrix}(SI)\\(DI)\end{Bmatrix}+\text{位移量}$$

3.3 堆栈与堆栈操作

堆栈是一种按特定顺序进行存取的存储区,这种特定顺序可归结为“后进先出(LIFO)”或“先进后出(FILO)”。在一般计算机中,堆栈主要用来暂存中断断点、子程序调用时的返回地址、状态标志及现场信息等,也可用于子程序调用时参数的传递。

3.3.1 堆栈结构

堆栈区通常是主存储器中指定的一个区域,也可以专门设置一个小而快的存储器作为堆栈区。在堆栈容量很小的情况下,还可以用一组寄存器来构成堆栈。

1. 寄存器堆栈

有些计算机中用一组专门的寄存器构成寄存器堆栈,又称为硬堆栈。这种堆栈的栈顶是固定的,寄存器组中各寄存器是相互连接的,它们之间具有对应位自动推移的功能,即可将一个寄存器的内容推移到相邻的另一个寄存器中,如图 3-14 所示。在执行压入操作(进栈)时,一个压入信号将使所有寄存器的内容依次向下推移一个位置,即寄存器 i 的内容被传送到 $i+1$,同时一个 n 位的数据被压入栈顶(寄存器 0)。在执行弹出操作(出栈)时,一个弹出信号将把所有寄存器的内容依次向上推移一个位置,即寄存器 i 的内容被传送到寄存器 $i-1$,栈顶(寄存器 0)的内容被弹出。

从图 3-14 可看出,上述堆栈中最多只能压入 k 个数据,否则将丢失信息。这种堆栈的工作过程很像子弹夹的弹仓,由于栈顶位置固定,故不必设置堆栈的栈顶指针。

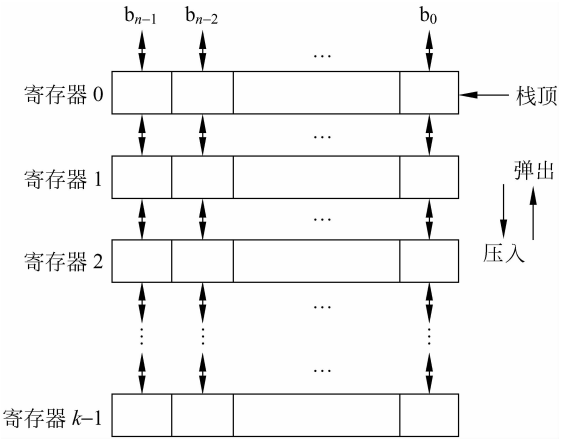


图 3-14　寄存器堆栈结构

2. 存储器堆栈

寄存器堆栈的成本比较高,不适于作大容量的堆栈,而从主存中划出一段区域来作为堆栈是最合算且最常用的方法。这种堆栈又称为软堆栈,堆栈的大小可变,栈底固定,栈顶浮动,故需要一个专门的硬件寄存器作为堆栈栈顶指针,简称栈指针(SP)。栈指针所指定的存储单元就是堆栈的栈顶。存储器堆栈又可分为两种:自底向上生成堆栈和自顶向下生成堆栈。假设栈指针始终指向栈顶的满单元,且压入和弹出的数据为一个字节。

(1) 自底向上生成(向低地址方向生成)堆栈

这种堆栈的栈底地址大于栈顶地址,如图 3-15 所示。因此,进栈时,堆栈指针 SP 的内容需要先自动减 1,然后再将数据压入堆栈;出栈时,需要先将堆栈中的数据弹出,然后 SP 的内容再自动加 1。进、出栈的过程可描述如下。

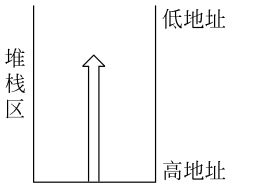


图 3-15　存储器堆栈结构

进栈:

(SP)－1→SP　　;修改栈指针
(A)→(SP)　　;将 A 中的内容压入栈顶单元

出栈:

((SP))→A　　;将栈顶单元内容弹出送入 A 中
(SP)＋1→SP　　;修改栈指针

其中,A 为寄存器或主存单元地址;(SP)表示堆栈指针的内容,即栈顶单元地址;((SP))表示栈顶单元的内容。

(2) 自顶向下生成(向高地址方向生成)堆栈

这种堆栈与自底向上堆栈正好相反,它的栈底地址小于栈顶地址。进栈时,先令(SP)＋1→SP,然后再压入数据;出栈时,先将数据弹出,然后(SP)－1→SP。

软堆栈的容量可以很大,而且可以在整个主存中浮动,但是速度比较慢,每访问一次堆栈实际就是访问一次主存。在一些大型的计算机系统中,希望堆栈的容量大、速度快,故将

前述两种堆栈组合起来构成软、硬结合的堆栈。在这样的堆栈中，一般压入、弹出操作在小容量的硬堆栈中进行，这样可保证访问速度快。当硬堆栈已满之后，每向硬堆栈压入一个数据，总是将其栈底寄存器中的数据压入软堆栈中，使堆栈总容量有效扩大；同样，数据出栈时，不断将软堆栈中栈顶的内容上移至硬堆栈的栈底寄存器中。显然它集中了硬堆栈速度快、软堆栈容量大的优点，只是在控制上稍复杂些，但这是完全可以实现的。

3.3.2 堆栈操作

堆栈操作既不是在堆栈中移动它所存储的内容，也不是把已存储在栈中的内容从栈中抹掉，而是通过调整堆栈指针而给出新的栈顶位置，以便对位于栈顶位置的数据进行操作。

在一般计算机中，堆栈主要用来暂存中断断点、子程序调用时的返回地址、状态标志及现场信息等，也可用于子程序调用时参数的传递，所以用于访问堆栈的指令只有进栈(压入)和出栈(弹出)两种。

在堆栈计算机(如 HP-3000 和 B5000 机等)中，算术逻辑类指令中没有地址码字段，故称为零地址指令。参加运算的两个操作数隐含地从堆栈顶部弹出，送到运算器中进行运算，运算的结果再隐含地压入堆栈。如果将算术表达式改写为逆波兰表达式，用零地址指令进行运算是十分方便的。例如，有算术表达式 $a\times b+c\div d$，运算结果送给 X，这个算术表达式可以用逆波兰法表示为 $ab\times cd\div +$。现在用零地址指令和一地址指令对该算式编程，并利用堆栈完成运算。假设堆栈采用自底向上生成方式，用大写字母 A 表示数据 a 的地址，其他依次类推，其程序段为：

```
PUSH A      ;数据 a 压入堆栈
PUSH B      ;数据 b 压入堆栈
MUL         ;完成 a×b
PUSH C      ;数据 c 压入堆栈
PUSH D      ;数据 d 压入堆栈
DIV         ;完成 c÷d
ADD         ;完成 a×b+c÷d
POP X       ;结果存入 X 单元
```

注意：执行一条零地址的双操作数运算指令，如果是软堆栈，则需要访问 4 次主存；如果是硬堆栈，则只需要访问一次主存。

3.4 指令类型

一台计算机的指令系统可以有上百条指令，这些指令按其功能可以分成几种类型，下面分别介绍。

3.4.1 数据传送类指令

数据传送类指令是最基本的指令类型，主要用于实现寄存器与寄存器之间、寄存器与主存单元之间以及两个主存单元之间的数据传送。数据传送类指令又可以细分为下列几种。

1. 一般传送指令

一般传送指令具有数据复制的性质,即数据从源地址传送到目的地址,而源地址中的内容保持不变。一般传送类指令常用助记符 MOV 表示,根据数据传送的源和目的的不同,又可分为以下几种传递方式:

① 主存单元之间的传送。

② 从主存单元传送到寄存器。在有些计算机中,该指令用助记符 LOAD(取数指令)表示。

③ 从寄存器传送到主存单元。在有些计算机里,该指令用助记符 STORE(存数指令)表示。

④ 寄存器之间的传送。

2. 堆栈操作指令

堆栈指令实际上是一种特殊的数据传送指令,分为进栈(PUSH)和出栈(POP)两种,在程序中它们往往是成对出现的。

如果堆栈是主存的一个特定区域,那么对堆栈的操作也就是对存储器的操作。

3. 数据交换指令

前述的传送都是单方向的。然而,数据传送也可以是双方向的,即将源操作数与目的操作数(一个字节或一个字)相互交换位置。

3.4.2 运算类指令

1. 算术运算类指令

算术运算指令主要用于定点和浮点运算。这类运算包括定点加、减、乘、除指令,浮点加、减、乘、除指令以及加 1、减 1、比较等,有些机器还有十进制算术运算指令。

绝大多数算术运算指令都会影响到状态标志位,通常的标志位有进位、溢出、全零、正负和奇偶等。

为了实现高精度的加减运算(双倍字长或多字长),低位字(字节)加法运算所产生的进位(或减法运算所产生的借位)都存放在进位标志中;在高位字(字节)加减运算时,应考虑低位字(字节)的进位(或借位),因此,指令系统中除去普通的加、减指令外,一般都设置了带进位加指令和带借位减指令。

2. 逻辑运算类指令

一般计算机都具有与、或、非和异或等逻辑运算指令。这类指令在没有设置专门的位操作指令的计算机中常用于对数据字(字节)中某些位(一位或多位)进行操作,常见的应用如下。

(1) 按位测(位检查)

利用“与”指令可以屏蔽掉数据字(字节)中的某些位。通常让被检查数作为目的操作

数，屏蔽字作为源操作数，要检测某些位，可使屏蔽字的相应位为“1”，其余位为“0”，然后执行“与”指令，则可取出所要检查的位来。

（2）按位清（位清除）

利用“与”指令还可以使目的操作数的某些位置“0”。只要源操作数的相应位为“0”，其余位为“1”，然后执行“与”指令即可。

（3）按位置（位设置）

利用“或”指令可以使目的操作数的某些位置“1”。只要源操作数的相应位为“1”，其余位为“0”，然后执行“或”指令即可。

（4）按位修改

利用“异或”指令可以修改目的操作数的某些位，只要源操作数的相应位为“1”，其余位为“0”，“异或”之后就达到了修改这些位的目的（因为 $A \oplus 1 = \overline{A}$，$A \oplus 0 = A$）。

（5）判符合

若两数相符合，其“异或”之后的结果必定为全“0”。

3. 移位类指令

移位指令分为算术移位、逻辑移位和循环移位 3 类，它们又可分为左移和右移两种。

（1）算术移位

算术移位的对象是带符号数，在移位过程中必须保持操作数的符号不变。当左移一位时，如不产生溢出，则数值乘以 2；而右移一位时，如不考虑因移出舍去的末位尾数，则数值除以 2，如图 3-16(a)所示。

（2）逻辑移位

逻辑移位的对象是无符号数，因此移位时不必考虑符号问题，如图 3-16(b)所示。从图中可以看出，逻辑左移指令和算术左移指令移位操作过程完全相同，这是因为正确的算术左移（不产生溢出时）与逻辑左移结果相同。

（3）循环移位

循环移位按是否与进位位一起循环又分为两种：小循环（不带进位循环），如图 3-16(c)所示；大循环（带进位循环），如图 3-16(d)所示。

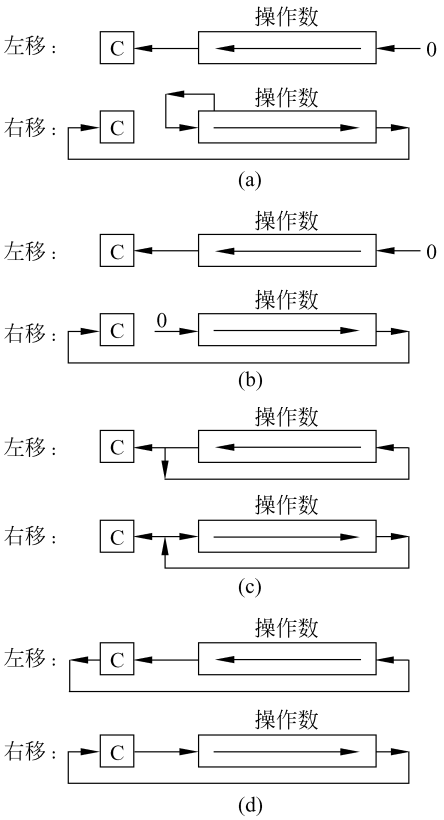


图 3-16　移位操作过程

3.4.3　程序控制类指令

程序控制类指令用于控制程序的执行顺序，并使程序具有测试、分析与判断的能力。因此，它们是指令系统中一组非常重要的指令，主要包括转移指令、子程序调用和返回指令等。

1. 转移指令

在程序执行过程中,通常采用转移指令来改变程序的执行顺序。转移指令又分无条件转移和条件转移两种:

① 无条件转移又称必转,它在执行时将改变程序的常规执行顺序,不受任何条件的约束,直接把程序转向该指令指出的新的位置并执行,其助记符一般为JMP。

② 条件转移必须受到条件的约束,若满足指令所规定条件,则程序转移;否则,程序仍顺序执行。条件转移指令主要用于程序的分支,当程序执行到某处时,要在两个分支中选择一支,这就需要根据某些测试条件做出判断。

无论是条件转移还是无条件转移都需要给出转移地址。若采用相对寻址方式,则转移地址为当前指令地址(即PC的值)和指令中给出的位移量之和,即(PC)+位移量→PC;若采用绝对寻址方式,则转移地址由指令的地址码字段直接给出,即A→PC。

条件转移指令采用相对寻址方式,通常位移量只有一个字节,这样转移范围只能在离当前PC的-128～+127个字节之内,在32位的80x86中,允许采用多字节表示位移量,此时转移范围可以超出原来的-128～+127。

转移的条件以某些标志位或这些标志位的逻辑运算作为依据,根据单个标志位的条件转移指令的转移条件是上次运算结果的某些标志,如进位标志、结果为零标志、结果溢出标志等,而用于无符号数和带符号数的条件转移指令的转移条件则是上述标志位逻辑运算的结果。

无符号数之间大小比较后的条件转移指令和带符号数之间的大小比较后的条件转移指令有很大不同。带符号数间的次序关系称为大于(G)、等于(E)和小于(L);无符号数间的次序关系称为高于(A)、等于(E)和低于(B)。

2. 子程序调用指令

子程序是一组可以公用的指令序列,只要知道子程序的入口地址就能调用它。通常把一些需要重复使用并能独立完成某种特定功能的程序单独编成子程序,在需要时由主程序调用它们,这样做既简化了程序设计,又节省了存储空间。

主程序和子程序是相对的概念,调用其他程序的程序是主程序,而被其他程序调用的程序是子程序。子程序允许嵌套,即程序A调用程序B,程序B又调用程序C,程序C再调用程序D……这个过程又称为多重转子。其中,程序B对于程序A来说是子程序,而程序B对于程序C来说是主程序。另外,子程序还允许自己调用自己,即子程序递归。

从主程序转向子程序的指令称为子程序调用指令,简称转子指令,其助记符一般为CALL。转子指令安排在主程序中需要调用子程序的地方,转子指令是一地址指令。

转子指令和转移指令都可以改变程序的执行顺序,但事实上两者存在着很大的差别:

① 转移指令使程序转移到新的地址后继续执行指令,不存在返回的问题,所以没有返回地址;而转子指令要考虑返回问题,所以必须以某种方式保存返回地址,以便返回时能找到原来的位置。

② 转移指令用于实现同一程序内的转移;而转子指令转去执行一段子程序,实现的是不同程序之间的转移。