第 1 章 计算机系统概述

计算机是一种能自动、快速、准确地实现信息存储、数值计算、数据处理和过程控制等多种功能的电子机器，是一个复杂的系统。由于其种类繁多且发展迅猛，因此，要清晰和完整地描述当代计算机系统的性质和特征必须关注计算机系统的一些基本概念，认识计算机系统的分层性质。本书采用自顶向下、由表及里的方法来描述计算机系统，从系统的主要部件开始描述它的结构和功能，然后逐级深入层次结构的底层。

本章首先简单介绍程序员与计算机的接口，即计算机的语言；然后介绍如何使用寄存器传输语言来描述数字系统，为后续描述计算机各功能部件作准备；接着介绍计算机的发展演变和性能指标；最后重点讨论计算机的基本组成和工作原理，以期提供一个计算机系统的整体概貌。

1.1 计算机的语言

计算机中的每一个动作或步骤都是按照程序来执行的。程序是计算机所执行指令的集合，是程序员根据所需完成的任务采用计算机语言来编写的，所以要控制计算机可以利用计算机语言向其发出命令。计算机语言是程序员与计算机之间的接口，一般地，计算机程序设计语言分为 3 类，即高级语言、汇编语言和机器语言。本节将讨论各种级别的程序设计语言及其相关性以及如何将其转换成可被处理器执行的形式，但并不深入研究任何细节。

1.1.1 高级语言

人类相互交流信息所用的语言称为自然语言，如汉语、英语、法语等，但目前计算机还不具备理解自然语言的能力，于是人们希望找到一种接近自然语言并能被计算机接受的语言，即计算机的高级语言。

高级语言(high-level language)是指那些具有最高级抽象的语言。这些语言隐藏了计算机和操作系统的细节,并与计算机隔离,称之为平台无关(platform independent),即同样的代码无需进行任何修改就可以转换并运行在具有不同处理器和操作系统的计算机上。例如,C++、Java 和 Fortran 等都是高级语言。

例如,对于表达式 A=B+C,使用高级语言来描述最简单、直观、高效且容易理解,如表 1-1 所示。然而,目前的计算机还不能直接执行用高级语言编写的程序,需要先将其翻译成机器所能执行的语言,即机器语言,再在计算机上执行。将高级语言翻译成机器语言的过程称为编译,完成该工作的系统软件称为编译器。

表 1-1 A=B+C 的三种可能表示

语言种类	表示形式
高级语言	A=B+C
汇编语言	ADD A,B,C
机器语言	1010 00 01 10

由于高级语言是平台无关的,因此同一高级语言的源代码可以经过编译后在不同的计算平台上运行。尽管理论上可以用一个单独的编译器产生适合于不同平台的目标代码,但实际上每个平台都有自己的编译器。此外,高级语言的语句功能相对强大,通常需要转换成多条机器代码指令,而且一条语句可能有多种有效的转换,从而使编译器的设计复杂化。

1.1.2 机器语言

机器语言(machine language)是处理器能直接识别并执行的唯一语言,其表现形式是二进制编码,见表 1-1。机器语言程序是机器指令的有序集合。机器指令通常由操作码和操作数两部分组成,操作码指出该指令所要完成的操作,即指令的功能,如 1010 表示加法等;操作数指出参与运算的对象(如 01、10 分别表示操作数 B 和 C)以及运算结果所存放的位置等(如 00 表示存放结果的 A)。

机器语言是最低级的程序设计语言,具有平台特定性,每一种处理器都有特定的机器语言。由于机器指令与处理器紧密相关,因此,不同种类的处理器所对应的机器指令也就不同,而且它们的指令系统往往相差很大。但对同一系列的处理器来说,为了满足各型号之间良好的兼容性,要做到新一代处理器的指令系统必须向下兼容先前同系列处理器的指令系统。只有这样,先前开发出来的各类程序在新一代处理器上才能正常运行。

用机器语言编写程序是早期经过严格训练的专业技术人员的工作,普通的程序员一般难以胜任,而且用机器语言编写的程序不易阅读、出错率高、难以维护,也不能直观反映使用计算机解决问题的基本思路,因此,现在几乎没有程序员采用机器语言来编写程序了。然而,由高级语言和汇编语言编写的程序都可以转换成机器语言,然后再由处理器执行。

1.1.3 汇编语言

汇编语言(assembly language)是面向机器的程序设计语言,它能利用计算机所有硬件

特性并能直接控制硬件。在汇编语言中，用助记符(mnemonic)代替机器指令的操作码，用地址符号(symbol)或标号(label)代替地址码，见表 1-1。其中，ADD 表示加法，A、B 和 C 分别表示不同的地址码。使用符号代替机器语言的二进制代码，就将机器语言变成了汇编语言，所以汇编语言也称符号语言。

汇编语言是较低级的程序设计语言，与机器语言类似，它不是平台无关的，每一种处理器都有它自己的汇编语言。以某种处理器的汇编语言编写的程序不能在另外一台具有不同处理器的计算机上运行。开发处理器的公司通常在设计新的处理器时都向下兼容以前的处理器。例如，Intel 公司的 Pentium Ⅲ处理器可以运行由 Pentium Ⅱ、Pentium Pro、Pentium、80486、80386、80286 和 8086 处理器的汇编语言编写的程序，即基本上可以运行任何为 IBM 兼容 PC 所写的代码。有了向下兼容特性，人们就可以购买具有当前先进工艺水平的新式计算机，而且可以使用老式计算机上的软件，从而节省成本和时间。

计算机不能直接识别使用汇编语言编写的程序，需要将汇编语言翻译成机器语言，这种起翻译作用的程序就是汇编程序，也叫汇编器。目前高级的汇编程序有 MASM、TASM 等。汇编程序把汇编语言翻译成机器语言的过程称为汇编。汇编程序通常仅运行在一种平台上，因此不需要针对不同平台的汇编程序。由于汇编语言采用符号来代替机器指令的二进制编码，而且助记符与指令代码一一对应，即每种汇编语言指令都唯一对应一种机器代码指令，因此，汇编语言的汇编器比高级语言的编译器简单得多。

汇编语言比机器语言易于读写、调试和修改，同时具有机器语言的全部优点，即目标代码简短，占用内存少，执行速度快，能有效地访问和控制计算机的各种硬件设备，如磁盘、存储器、CPU、I/O 端口等。但在编写复杂程序时，相对于高级语言来说代码量较大，开发效率较低，而且汇编语言依赖于具体的处理器体系结构，不能直接在不同的处理器体系结构之间移植。因此，目前汇编语言主要用于编写系统程序和过程控制软件。

1.2　寄存器传输语言

在计算机的各大部件中都存在大量的硬件寄存器，通过对这些寄存器内容的传送和加工处理，就可以达到对数据的控制、运算、存取和输入/输出的目的。在描述计算机的工作原理之前，作为其基础，先简单介绍一种硬件设计描述语言，即 RTL。

1.2.1　微操作和寄存器传输语言

RTL(register transfer language，寄存器传输语言)是初级硬件描述语言，适用于描述寄存器级的硬件组成，能精确而简练地描述计算机的各种基本操作，对于讨论计算机的硬件组成和分析计算机各部件的工作原理是很方便的。任何一个计算机都可以看成一个复杂的时序数字系统，在系统中有控制流和控制流下的数据操作，因此，可将计算机系统分为数据部分和控制部分，其中，数据部分由寄存器、寄存器输入/输出线路、数据输入/输出线路组成；控制部分由寄存器、组合逻辑线路、控制输入/输出线路组成。由这两部分可以看出，寄存器

的描述、寄存器信息传送的描述、各种控制条件的描述等是系统描述的最基本要素。RTL是完成上述各种描述的有力工具。

微操作(micro-operation)是计算机中最基本的操作,这些操作可以简单到从一个寄存器复制数据到另一个寄存器中,或者更复杂,例如,把两个寄存器中的数据相加并存储到第三个寄存器中。设计时序数字系统时,设计者可首先用微操作表述系统的行为,之后设计硬件来匹配这些表述。

RTL所用的基本符号、控制条件符号以及它们所表示的传输语句如表1-2所示。

表1-2 RTL基本传输语句和控制条件的符号

类 型	符号表示	说 明
基本符号	大写字母和数字	表示一个寄存器
	下角标	表示寄存器的一位
	括号( )	表示寄存器的一部分
	箭头 ←	表示信息的传送
	冒号:	表示控制条件的结束
	逗号,	分隔列出同时执行的微操作
寄存器传送	A←B	B寄存器内容传送到A寄存器
	MAR←MBR(AD)	MBR中地址部分内容传送到MAR
	A←Const	二进制常数传送到A寄存器
	ABUS←R_1 R_2←ABUS	将R_1内容传送到总线A,同时将总线A的内容再传送到R_2
	MBR←M	存储器读出由MAR规定的存储字M的内容到MBR
	M←MBR	将MBR内容写入MAR规定的存储字M中去
控制条件	+	表示逻辑“或”运算
	×	表示逻辑“与”运算(或不用算符)
	/A	在控制条件中表示一个变量的“反”
	T_1X+T_2Y:	表示控制条件,以“:”结尾
	IF (C=0) THEN (A←B)	控制条件语句,控制条件在IF后面的括号内,THEN后面的括号内为微操作

用RTL描述微操作有以下三种语句形式:

条件:微操作

条件:IF (另一个控制条件) THEN(微操作)

IF (整个控制条件) THEN (微操作)

例如,有一个包含两个1位寄存器X和Y的数字系统,在输入控制α为高电平时,复制寄存器Y的内容到寄存器X中的微操作可以表示为

α: X←Y

上述微操作没有指出数据怎样从Y复制到X,它仅仅规定了要执行的传送。此微操作

可由直接连接实现，如图 1-1(a)所示；或通过总线连接实现，如图 1-1(b)所示。两者都是有效的微操作实现方式，在系统设计中选用哪种实现方法一般由系统设计者决定。

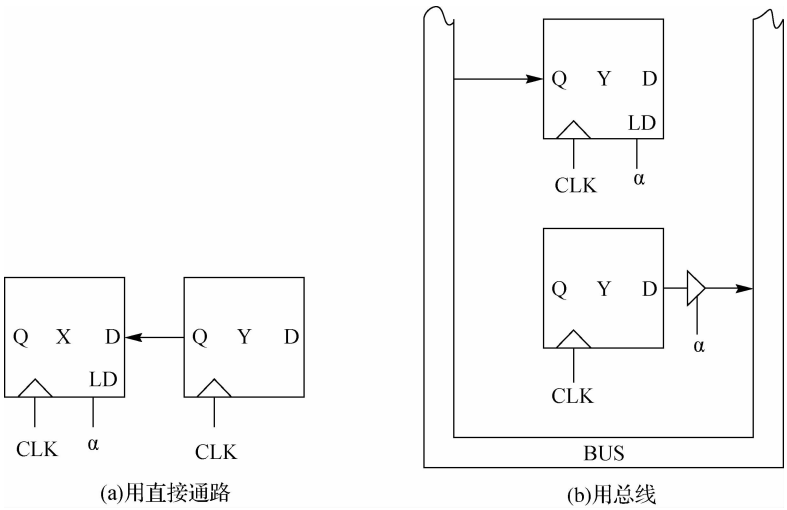


图 1-1　数据传送 α：X←Y 的实现

提高系统性能的一种方法是同时执行两个或多个微操作。在这种表示法中，微操作之间用逗号隔开；它们书写的顺序无关紧要，因为它们是并行执行的。例如，如果某系统在 α＝1 时执行 X←Y 和 Y←Z 的传送，那么这种情况可以表示成

$$\alpha: X \leftarrow Y, Y \leftarrow Z$$

或者

$$\alpha: Y \leftarrow Z, X \leftarrow Y$$

注意，同时从寄存器中读出和写入是可能的。从 Y 中读出的值是 Y 先前的值，不是 Y 从 Z 中装载的值。例如，如果正好在 α 变成 1 之前，X＝0、Y＝1 并且 Z＝0，那么上述微操作将置 X＝1，Y＝0。这对移位操作是特别有用的。图 1-2 给出了实现上述微操作的硬件连接。注意，单一的总线不能用在这里，因为一根总线在某一时刻只能保持一个值。当 α＝1 时，Y 和 Z 必须同时在数据通路上传输。

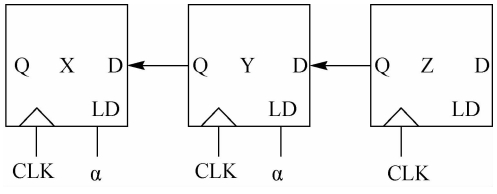


图 1-2　数据传送 α：X←Y，Y←Z 的实现

例如，当一位教授在黑板上写字时，所有学生都可能阅读他书写的文字。数字系统中的寄存器与此类似，有时需要同时复制相同的数据到多个目的地。考虑 α＝1 时发生下述情况：

$$\alpha: X \leftarrow Y, Z \leftarrow Y$$

寄存器 Y 可同时被多个寄存器读取，两个微操作可同时执行。其中一种实现方法如

图 1-3 所示。

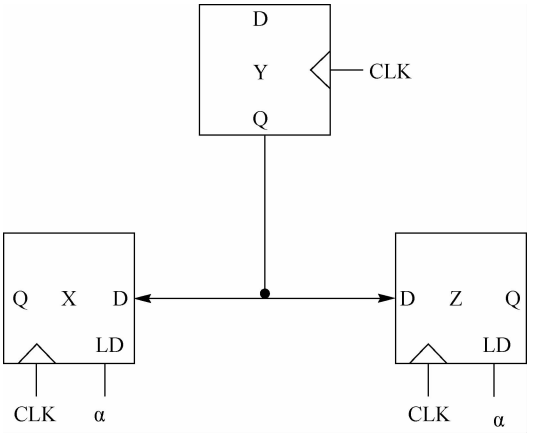


图 1-3 实现数据传送 α: X←Y,Z←Y

另一方面,多个教授不可能同时在黑板的相同位置写字,这将使书写变得混淆和没有意义。类似地,数字系统不能同时向同一寄存器中写入两个不同的值。例如,α: X←Y, X←Z 是无效的。

有时需要访问一个寄存器的单一位或位组。单一位可以用带有下标的字母引用,如 X_3 或 Y_2 等。位组在 RTL 中可以用一个域(range)引用,它们包含在圆括号中,如 X_3 和 X_1 可以写成 X(3-1)或 X(3:1)。于是单一位或位组就可以像整个寄存器的使用一样在微操作中使用。下述表示是有效的:

$$\alpha: X(3\text{-}1) \leftarrow Y(2\text{-}0)$$

$$\beta: X_3 \leftarrow X_2$$

$$\gamma: X(3\text{-}0) \leftarrow X(2\text{-}0), X_3$$

需要特别注意最后一个表达式,由于不可能把 X_2、X_1、X_0 和 X_3 按顺序表示为某个连续的域,因此,可以用逗号分隔的子域来表示。例如,表达式 X(2-0,3)即为此情况,它实现了循环左移传送。

到现在为止,提及的每个微操作仅是从一个寄存器传送数据到另外一个寄存器,实际上也可以装载一个规定的常量到寄存器中。还有一些微操作执行数据的算术运算、逻辑运算和移位运算。这些描述语言均非常简单,在此不再赘述。

1.2.2 用 RTL 表示数字系统

寄存器传输语言可以用来表示从简单到复杂的任意时序数字系统的行为。

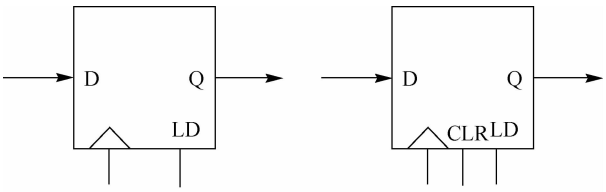
1)数字元件表示

考虑如图 1-4(a)所示的 D 触发器。其功能可用 RTL 语句表示为

$$LD: Q \leftarrow D$$

当 LD 的输入为高电平时,装载输入端 D 的值,并且输出端 Q 可得到此值。这仅发生在时钟的上升沿由 0 转变为 1 时。在所有的 RTL 代码中,为了保证微操作的执行,除状态条

件之外，还要求时钟有正确的跳变或电平。



(a)无清0输入端　　(b)有清0输入端

图 1-4　D 触发器

图 1-4(b)所示 D 触发器有一个同步清 0 输入端。当 CLR＝1 时，触发器应被置为 0。用 RTL 代码描述此触发器为

$$\text{LD：}Q\leftarrow D$$
$$\text{CLR：}Q\leftarrow 0$$

然而当 D、LD 和 CLR 都等于 1 时，系统会失败。因为 LD＝1，第一个表达式将置 Q 为 1；同时，CLR＝1 会执行第二个表达式，即置 Q 为 0。两者同时置 Q 为不同的值，这显然是不可能的。解决的方法是改变条件使两者互斥。下述两种方法均有效，前者让 CLR 输入端优先，而后者让 LD 优先。

$$\text{CLR}'\text{LD：}Q\leftarrow D \qquad \text{LD：}Q\leftarrow D$$
$$\text{CLR：}Q\leftarrow 0 \qquad \text{LD}'\text{CLR：}Q\leftarrow 0$$

在上述 RTL 代码中，注意 RTL 语句 CLR′LD：Q←D 中的组合条件。为了执行微操作，可能要求满足多个条件。这里仅当 CLR＝0 和 LD＝1 同时满足时，才执行 Q←D。

第二个例子，考虑一个没有 CLR 输入端的 JK 触发器。它的行为可用 RTL 语句描述如下：

$$\text{J}'\text{K：}Q\leftarrow 0$$
$$\text{JK}'\text{：}Q\leftarrow 1$$
$$\text{JK：}Q\leftarrow Q'$$

当 J＝K＝0 时，不满足条件，触发器将保持原值，此时不需要 RTL 语句，因为没有传送。

2)数字系统表示

用 RTL 表示单一元件并不是特别有用，RTL 的最大用途在于表示一个完整系统的行为，它独立于实现此系统的元件。

下面介绍如何使用 RTL 来设计一个模 6 计数器。首先，用 RTL 表示计数器的功能，之后用数字逻辑实现 RTL 代码。模 6 计数器是一个 3 位计数器，它按如下顺序计数：

$$000\rightarrow 001\rightarrow 010\rightarrow 011\rightarrow 100\rightarrow 101\rightarrow 000\rightarrow\cdots(0\rightarrow 1\rightarrow 2\rightarrow 3\rightarrow 4\rightarrow 5\rightarrow 0\cdots)$$

其输入端 U 控制计数。当 U＝1 时，计数器在时钟的上升沿增加它的值；当 U＝0 时，不管时钟的值如何，它都保持当前值不变。计数器的值用 3 位输出 $V_2V_1V_0$ 表示，当值从 5 变为 0 时，进位输出 C 的值为 1，否则为 0。在该例子中，C 值保持 1 不变，直到计数器从 0 变为 1 为止。

该计数器的有限状态机需有 6 个状态，任意标识为 S_0，S_1，S_2，S_3，S_4 和 S_5，状态 S_i 相应于计数器的输出 i，状态按如下顺序排列：

$$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_4 \rightarrow S_5 \rightarrow S_0 \rightarrow \cdots$$

此外，为了处理模 6 计数器电源开启处于无效状态的情形，包含了另外两个状态，即 S_6 和 S_7。根据该计数器的工作情况得出其状态表如表 1-3 所示，其状态图如图 1-5 所示。

表 1-3　模 6 计数器的状态表

当前状态	U	下一状态	C	$V_2V_1V_0$
S_0	0	S_0	1	000
S_0	1	S_1	0	001
S_1	0	S_1	0	001
S_1	1	S_2	0	010
S_2	0	S_2	0	010
S_2	1	S_3	0	011
S_3	0	S_3	0	011
S_3	1	S_4	0	100
S_4	0	S_4	0	100
S_4	1	S_5	0	101
S_5	0	S_5	0	101
S_5	1	S_0	1	000
S_6	×	S_0	1	111
S_7	×	S_0	1	111

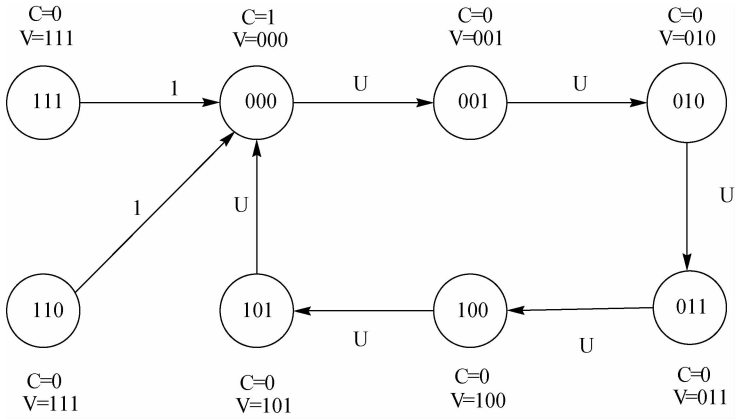


图 1-5　模 6 计数器的状态图

为了用 RTL 表示该系统，首先需定义 $S_0 \sim S_7$ 的组合条件，对应于 $V_2V_1V_0$ 从 000～111；然后考虑模 6 计数器的各种可能行为。当计数器的值从 000 变为 100 且它的 U 信号有效时，计数器的输出增加，相应条件为$(S_0+S_1+S_2+S_3+S_4)U$。在这种情形下，C 也被置为 0。用 V 表示值 $V_2V_1V_0$，则这两个行为可表示为 V←V+1，C←0。相应的 RTL 语句如下所示（“+”在“:”左边表示逻辑或，在右边则表示算术加）：

$$(S_0+S_1+S_2+S_3+S_4)U: V \leftarrow V+1, C \leftarrow 0$$

此外，当计数器在状态 S_5（V＝101）且 U＝1 时，计数器一定置为 000，且 C 置为 1，即

$$S_5U：V\leftarrow 0，C\leftarrow 1$$

在无效状态时，不管 U 值为多少，均发生如下赋值：

$$S_6+S_7：V\leftarrow 0，C\leftarrow 1$$

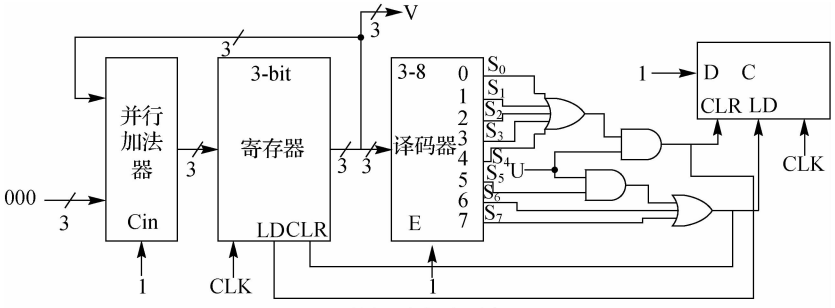
现在仅留下一种条件$(S_0+S_1+S_2+S_3+S_4+S_5)U'$没有考虑。当计数器在有效状态且 U＝0 时，此条件为真。在这种条件下，计数器保持当前值与 C 值不变。因为没有数据传送发生，所以此种情况不需要用 RTL 语句表示。在 RTL 中，当没有条件满足时，传送不会发生。

既然 S_5U 和 S_6+S_7 触发相同的微操作，那么可以把它们结合起来。因此，整个模 6 计数器的行为可以用以下两条 RTL 语句来表示：

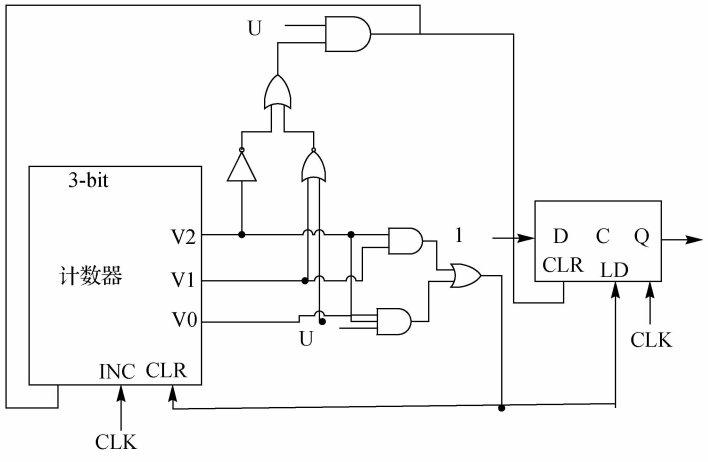
$$(S_0+S_1+S_2+S_3+S_4)U：V\leftarrow V+1，C\leftarrow 0$$

$$S_5U+S_6+S_7：V\leftarrow 0，C\leftarrow 1$$

图 1-6 所示为该 RTL 代码的两种实现方式。图 1-6(a)利用一个 3 位并行加法器单独产生 V＋1 值和置 C 值，其中译码器转换 $V_2V_1V_0$ 到状态值 S_0～S_7；图 1-6(b)利用的是一个 3 位计数器，当它从 101～000 计数或从无效状态重置时，置 C 为 1。



(a)利用一个加法器实现



(b)利用一个计数器实现

图 1-6　模 6 计数器 RTL 代码的两种实现方式

1.3 计算机的发展演变和性能

计算机的发展历史主要由提高处理器速度、减小部件尺寸、增大存储容量、加快 I/O 能力和速度等来表征。而计算机系统性能设计的一个关键问题是各元器件之间的性能平衡，以使在一个领域内所获得的性能增益不被另一领域的滞后所阻碍。

1.3.1 计算机简史

自从 1946 年美国宾夕法尼亚大学的 W·Mauchly 和 J·P·Eckert 两位教授研制的第一台计算机 ENIAC(electronic numerical integrator and calculator)问世以来，计算机的发展已经历了 60 多年。如何划分各计算机时代，目前意见不太一致，但基于计算机所采用的基本硬件技术来划分，已成为人们的一个共识，如表 1-4 所示。新一代计算机以比旧一代计算机具有更快的速度、更大的存储容量和更小的尺寸为特征。

表 1-4 计算机的发展阶段

发展阶段	大致时间	技　术	典型速度/(次/秒)
1	1946～1959 年	电子管	4 万
2	1960～1964 年	晶体管	20 万
3	1965～1970 年	小规模和中规模集成电路	100 万
4	1971 年至今	大规模/超大规模集成电路	1 000 万

1)第一代(1946～1959 年):电子管计算机

主要特点:逻辑元件 —— 电子管。

主　　存 —— 磁鼓。

辅　　存 —— 磁带。

软　　件 —— 机器语言、符号语言。

应　　用 —— 科学计算。

主要成就:

(1)数字电子计算机的出现揭开了人类历史的新篇章。

(2)1945 年，美国数学家、普林斯顿大学教授、ENIAC 项目顾问约翰·冯·诺依曼(John Von Neumann)在一份新型计算机 EDVAC(electronic discrete variable computer，电子离散变量计算机)的计划中提出了存储程序的概念，即程序和数据一起存放在存储器中。该思想奠定了现代计算机组成与工作原理的基础，被称为冯·诺依曼思想。以此思想为基础的各类计算机被称为冯·诺依曼计算机。

典型代表有 ENIAC、IAS 计算机、UNIVAC 和 IBM 701 计算机。

2)第二代(1960～1964 年):晶体管计算机

主要特点:逻辑元件 —— 晶体管。

主　　存 —— 磁芯。

辅　　存 —— 磁盘。

软　　件 —— 高级程序设计语言、操作系统。

应　　用 —— 除科学计算外,已应用于数据处理、过程控制。

主要成就:

(1)首次将晶体管用于计算机,克服了第一台计算机体积大、造价高、功耗大和不稳定等缺陷。晶体管是 1947 年由以贝尔实验室的 W. Shockly,J. Bardeen 和 W. H. Brattain 为核心的固体物理研究小组研制成功的。

(2)发明了高级语言。1956 年,美国国防部发明了第一个专用的高级语言,即 Ada 语言。1957 年,IBM 公司的 Backus 发明了 FORTRAN 高级语言,主要用于科学计算。

(3)计算机兼容问题的产生。其包括硬件兼容和软件兼容。

典型代表有 IBM 7000 系列计算机和 PDP-1 小型计算机。

3)第三代(1965～1970 年):集成电路计算机

主要特点:逻辑元件 —— 集成电路(IC)。

主　　存 —— 半导体。

辅　　存 —— 磁盘。

软　　件 —— 高级程序设计语言、操作系统。

应　　用 —— 科学计算、数据处理、过程控制。

主要成就:

(1)IBM 公司首次提出了系列机的概念,圆满地解决了计算机的兼容问题。

(2)利用微程序控制技术使控制器的设计规整化。

(3)结构化程序设计思想成熟,软、硬件设计标准化。

典型代表是 IBM 360 系列机和 DEC PDP-8 小型机。

4)第四代(1971 年至今):大规模/超大规模集成电路计算机

主要特点:逻辑元件 —— 大规模/超大规模集成电路(LSI/VLSI)。

主　　存 —— LSI/VLSI 半导体芯片。

辅　　存 —— 磁盘、光盘。

软　　件 —— 高级程序设计语言、操作系统。

应　　用 —— 科学计算、数据处理、过程控制,并进入以计算机网络为特征的应用时代。

主要成就:

(1)1971 年,美国 Intel 公司成功研制出了 Intel 4004 微处理器芯片。从此,随着 LSI/VLSI 技术的发展,微处理器每隔几年就有一个新的产品问世,至今已发展到奔腾处理器和酷睿多核处理器。表 1-5～表 1-8 给出了 Intel 微处理器的演变历史。

(2)微型计算机的出现,1981 年,IBM 公司正式推出了全球第一台个人计算机 IBM PC。

(3)面向对象、可视化程序设计概念出现;软件产业高度发达,各种实用软件层出不穷,极大地方便了用户。

(4)计算机技术与通信技术相结合的计算机网络把世界紧密地联系在一起。

(5)多媒体技术的兴起。

典型代表是 Intel x86 系列计算机、IBM PC 和各种超级计算机。

表 1-5　20 世纪 70 年代的处理器

型　　号	发布时间	时钟频率	总线宽度	晶体管数量	特征尺寸/μm	可寻址存储器
4004	1971 年	108 kHz	4 位	2 300	10	640 B
8008	1972 年	108 kHz	8 位	3 500	—	16 KB
8080	1974 年	2 MHz	8 位	6 000	6	64 KB
8086	1978 年	5 MHz, 8 MHz, 10 MHz	16 位	29 000	3	1 MB
8088	1979 年	5 MHz, 8 MHz	8 位	29 000	6	1 MB

表 1-6　20 世纪 80 年代的处理器

型　　号	发布时间	时钟频率	总线宽度	晶体管数量	特征尺寸/μm	可寻址存储器	虚拟存储器	高速缓存
80286	1982 年	6～12.5 MHz	16 位	134 000	1.5	16 MB	1 GB	—
386TM DX	1985 年	16～33 MHz	32 位	275 000	1	4 GB	64 TB	—
386TM SX	1988 年	16～33 MHz	16 位	275 000	1	16 MB	64 TB	—
486TM DX	1989 年	25～50 MHz	32 位	1 200 000	0.8～1	4 GB	64 TB	8 KB

表 1-7　20 世纪 90 年代的处理器

型　　号	发布时间	时钟频率	总线宽度	晶体管数量	特征尺寸/μm	可寻址存储器	虚拟存储器	高速缓存
Pentium	1993 年	60～166 MHz	32 位	3.1 百万	0.8	4 GB	64 TB	8 kB
Pentium Pro	1995 年	150～200 MHz	64 位	5.5 百万	0.6	64 GB	64 TB	512 KB L1 和 1 MB L2
Pentium Ⅱ	1997 年	200～300 MHz	64 位	7.5 百万	0.35	64 GB	64 TB	512 KB L2
Pentium Ⅲ	1999 年	450～600 MHz	64 位	9.5 百万	0.25	64 GB	64 TB	512 KB L2

表 1-8　最近的处理器

型　　号	发布时间	时钟频率	总线宽度	晶体管数量	特征尺寸/μm	可寻址存储器	虚拟存储器	高速缓存
Pentium4	2000 年	1.3～1.8 GHz	64 位	4 200 万	0.18	64 GB	64 TB	256 KB L2
Core 2Duo	2006 年	1.06～1.2 GHz	64 位	1.67 亿	0.065	64 GB	64 TB	2 MB L2
Core 2Quad	2008 年	3 GHz	64 位	8.2 亿	0.045	64 GB	64 TB	6 MB L2
Core i7 980x	2010 年	3.33 GHz	64 位	11.7 亿	0.032	64 GB	64 TB	12 MB L3

如果说第二代电子计算机是由于晶体管代替电子管所引起的飞跃，那么第三代、第四代电子计算机的产生则是由制造工艺的革新所引起的。

20 世纪 80 年代初以来，许多科学家一直在预测第五代计算机将朝哪个方向发展，综合起来大概有以下几个研究方向：

- 人工智能计算机。
- 超级计算机。
- 激光计算机。
- 超导计算机。
- 生物晶体计算机(DNA 计算机)。
- 量子计算机。

5)嵌入式系统和 ARM

随着计算机技术和产品对其他行业的广泛渗透，20 世纪 70 年末诞生了一种有别于通用计算机的系统，即嵌入式系统。通用计算机一般具有标准的硬件配置，通过安装不同的应用软件，以适应各种不同的应用需求；而嵌入式系统一般是以应用为中心，以计算机技术为基础，软、硬件可裁剪，适应应用系统对功能、可靠性、成本、体积以及功耗严格要求的专用计算机系统。在许多情况下，嵌入式系统是大型产品和系统的组成部分，如汽车的刹车系统等。嵌入式系统一般由嵌入式微处理器、外围硬件设备、嵌入式操作系统以及用户的应用程序等 4 个部分组成，用于实现对其他设备的控制、监视或管理等功能。

嵌入式系统经历了单片微型计算机(single chip microcomputer，SCM)、微控制器(micro controller unit，MCU)和片上系统(system on chips，SoC)三大阶段，目前正处于以 Internet 为标志的发展阶段。

ARM 是一种由英国剑桥 ARM 公司设计的基于 RISC 的微处理器和微控制器序列。该公司并不生产处理器，而是设计微处理器和多核的体系结构，然后向制造商发放许可。ARM 芯片是高速的处理器，由于它们的小特征尺寸和低能耗需求，因此被广泛应用于 PDA 以及其他便携设备中，包括手机、游戏机以及各种消费产品等。ARM 是迄今为止世界上各种应用中使用最广泛的处理器体系结构。

ARM 技术的起源可以追溯到英国的 Acorn 计算机公司。20 世纪 80 年代早期，Acorn 公司获得了英国广播公司(BBC)的合同，负责为 BBC 的计算机文化项目开发一款新的微计算机体系结构。这个合同的成功促使 Acorn 公司继续开发出了其第一款商用 RISC 商业处理器，即 ARM(Acorn RISC machine)。1985 年推出的第一个版本 ARM1，主要用于内部研

究和开发。同年推出了 ARM2，它在相同的物理空间内比 ARM1 具有更强的功能和更快的速度。1989 年推出了 ARM3。

这个时期的 Acorn 公司利用 VLSI 技术进行处理器芯片的实际制造，同时授权其他公司在其产品中使用 ARM 也获得了一些成功，尤其是在嵌入式处理器中。

ARM 设计顺应了嵌入式应用中对高性能、低功耗、小体积和低成本处理器不断增长的商业化需求，但进一步发展超出了 Acorn 的能力范围，于是，由 Acorn、VLSI 以及苹果计算机公司作为股东，成立了一家新公司，即 ARM 有限公司。Acorn 的 RISC 机器变成了先进的 RISC 计算机。新公司最先推出了 ARM6，它是 ARM3 的一种改进版本，接着，公司推出了许多新的系列，以增强相关功能和性能。表 1-9 列出了各种 ARM 结构系列的一些特征。

表 1-9　ARM 的进展

序　　列	显著特征	cache	典型的 MIPS@MHz
ARM1	32 位 RISC	无	—
ARM2	乘法和交换指令；集成存储器管理单元、图形和 I/O 处理器	无	7 MIPS@12 MHz
ARM3	第一次使用处理器 cache	4 KB 统一	12 MIPS@25 MHz
ARM6	第一次支持 32 位地址；浮点单元	4 KB 统一	28 MIPS@33 MHz
ARM7	集成 SoC	8 KB 统一	60 MIPS@60 MHz
ARM8	5 段流水线；静态分支预测	8 KB 统一	84 MIPS@72 MHz
ARM9	—	16 KB/16 KB	300 MIPS@300 MHz
ARM9E	增强 DSP 指令	16 KB/16 KB	220 MIPS@200 MHz
ARM10E	6 段流水线	32 KB/32 KB	—
ARM11	10 段流水线	可变的	740 MIPS@665 MHz
Cortex	13 段超级流水线	可变的	2 000 MIPS@1 GHz
XScale	应用型处理器；7 段流水线	32 KB/32 KB L1 512 KB L2	1 000 MIPS@1. 25 Hz

ARM 处理器被设计用来满足以下三种系统类别的需要：

- 嵌入式实时系统。存储、汽车和动力火车、工业以及网络应用的系统。
- 应用平台。在无线消费娱乐和数字图像应用领域中，运行开放式操作系统的设备，开放式操作系统包括 Linux、Palm OS、Symbian OS 和 Windows CE。
- 安全应用。智能卡、SIM 卡和支付终端。

1.3.2　性能指标

在评价已有系统和设置新系统的需求时，性能是必须考虑的关键因素之一。一台计算机性能高低的好坏受其系统结构、硬件组成、外设配置、软件种类等多个因素影响。早期评价计算机，多局限于字长、速度和容量，实际应用表明，只考虑上述三大因素是远远不够的。

现在，应予以考虑的因素更多、更全面，只有综合各项指标，才能正确评价与选择计算机系统。

另外，不同的用户注重不同的性能指标。例如，科学计算用户最关心的是运算速度和精度；军事用户最关心的是精度、可靠性和环境的适应性；过程控制用户最关心的是实时性和可靠性；数据处理用户最关心的是存储容量；维护人员最关心的是可用性和可维性等。下面就介绍一些常用的性能参数。

1)主频

主频或时钟周期是计算机的主要性能指标之一，它在很大程度上决定了计算机的运行速度。CPU 的工作节拍是由主时钟控制的，主时钟不断产生固定频率的时钟脉冲，这个主时钟的频率就是 CPU 的主频。主频越高，CPU 的工作节拍就越快，运算速度就越快。主频通常用一秒钟内处理器所能发出的电子脉冲数来测定，计量单位一般为兆赫兹(MHz)。

由于一条指令的执行包含很多离散的步骤，例如，取指令、译码指令、存/取数据和执行运算等，因此，大多数处理器的大部分指令需要多个时钟周期才能完成。有些指令可能只需要几个周期，而另一些指令则需要几十个周期。此外，当使用流水线时，多条指令被同时执行，因此，不同处理器时钟速度的直接比较并不能说明性能的整体情况。

2)运算速度

运算速度是计算机工作能力和生产效率的主要表征，它取决于在给定的时间内处理器所能处理的数据量和处理器的时钟频率。通常用每秒执行指令的条数来表示，其计量单位为百万条指令每秒(MIPS)和百万次浮点运算每秒(MFLOPS)。MIPS 用来描述计算机的定点运算速度；MFLOPS 则用来描述计算机的浮点运算速度，表征计算机的科学计算性能。

3)运算精度

运算精度也是计算机的重要性能指标之一，科学计算类计算机更是如此。运算精度通常以计算机处理信息时能直接处理的二进制信息位数来定义。这个位数通常与计算机 CPU 中存储数据的寄存器的位数相同，位数越多，精度越高。参与运算的数的基本位数通常用基本字长表示，因此，字长也在一定情况下标志着计算精度。基本字长决定着寄存器、加法器、数据总线等的位数，直接影响着硬件的代价。为了适应不同类型计算的需要，并较好地协调精度与造价的关系，许多计算机允许变字长计算，如半字长、全字长、双字长等。早期的微型计算机字长多为 4 位、8 位和 16 位，现在多以 32 位和 64 位为主。

4)主存容量

主存储器用来存储数据和程序，直接与 CPU 交换信息。主存的容量越大，可存储的数据和程序就越多，处理问题的能力也就越强；而且与外存储器的信息交换次数越少，系统的效率就越高。因此，主存容量是衡量计算机性能的指标之一。以字为单位的计算机常用字数乘以字长表示主存容量，如 4 K×16 位，表示主存有 4 096 个单元，每个单元字长为 16 位。以字节(B)为单位的计算机则以字节数表示主存容量，例如，32 KB 表示主存容量为 32 768×8 位。

5)存取周期

主存进行一次完整的读/写操作所需的时间，即主存进行连续读/写操作所允许的最小

时间间隔，称为存取周期，它包括读出时间和将读出信息重新写入原单元（在破坏性读出时）所需的全部时间。因此，存取周期既是表征主存性能的基本参数，也是反映计算机整机特性的重要参数。显然，存取周期越短，表明从主存存取信息的时间越短，计算机系统的性能越高。

6）系统配置

硬件配置主要指外部设备的配置情况，包括系统允许配置外设的最大数量、种类及输入/输出能力，基本配置外设的种类、数量、性能、加速度及分辨率等。

软件配置主要指操作系统的种类和版本，它表明其功能的强弱。另外，系统配置的工具软件及其他支持软件和应用软件的种类，界面是否友好，是否有数据库管理系统等都是软件配置应考虑的因素。

7）RASIS 特性

可靠性（reliability）、可用性（availability）、可维性（serviceability）、完整性（integrality）和安全性（security）统称 RASIS 特性，它们是衡量计算机系统性能的五大功能特性。

可靠性无疑是十分重要的指标，它表示计算机系统在规定的工作条件下和预定的工作时间内持续正确运行的概率。可靠性一般用平均无故障时间或平均故障间隔时间 MTBF（mean time between failures）来衡量。MTBF 越大，系统可靠性越高。

可维性指系统发生故障后能尽快修复的能力，一般用平均修复时间 MTTR（mean time to repair）来表示。MTTR 越小，表明系统的可维护性越好。

一个运行的系统不可能完全避免故障的发生，但希望修复的时间短，这样可供利用的时间就长。系统可靠性越高，可用性就越好。

8）兼容性

兼容性（compatibility）是指一个系统的硬件或软件与另一个系统或多种系统的硬件或软件的兼容能力，是指系统间某些方面具有的并存性，即两个系统之间存在一定程度的通用性。兼容是广泛的概念，它包括数据和文件的兼容，程序和语言的兼容，系统程序的兼容，设备的兼容，以及向上兼容等。兼容可使机器承前启后，便于推广，也可减少工作量。因而这也是用户通常要考虑的特性之一。

9）功耗

随着芯片上逻辑密度和时钟速度的提高，芯片消耗的功率密度也随之提高。高密、高速芯片的散热困难已成为一个重要的设计问题。

10）性能价格比

性能是指机器的综合性能，包括硬件、软件的各种性能。价格不但指主机，也包括整个系统的价格。显然，性能价格的比值越大越好，它是客户对经济效益的选择依据之一。

除上述性能指标外，还有其他性能指标，例如，综合性指标包括吞吐率、响应时间、利用率；定性指标包括保密性、可扩充性；功能特性包括汉字处理能力、联机事务处理能力、I/O 总线特性、网络功能等。总之，计算机性能评价是比较复杂和细致的工作。

1.4　计算机的基本组成

计算机系统由硬件和软件两部分组成。硬件是构成计算机系统的设备实体，包括运算器、控制器、存储器、输入设备和输出设备等五大部件。软件包括各类程序和文件，分为系统软件和应用软件。

硬件是躯体，是物质基础；软件是智慧，是灵魂，是硬件功能的完善与扩充。没有硬件或者没有良好的硬件，运行软件就无从谈起，也就无法计算、处理某一方面的问题。没有软件或者没有优秀的软件，计算机就是一个空壳，根本无法工作或者不能高效率地工作。因此，只有硬件与软件有机组合才能构成一个实用的计算机系统。

1.4.1　冯·诺依曼体系结构

1945 年 ENIAC 项目顾问、数学家约翰·冯·诺依曼在一份新型计算机 EDVAC 的计划中提出了存储程序的思想，几乎与此同时，阿兰·图灵也提出了同样的构想。

1946 年冯·诺依曼和他的同事们在普林斯顿高级研究院开始设计一种新的存储程序计算机，即 IAS 计算机。虽然直到 1952 年仍未完成，但它却成为了后来通用计算机的原型。以后凡以此思想为基础的各类计算机都称为冯·诺依曼计算机。其特点可以归纳如下：

(1)计算机由运算器、控制器、存储器、输入设备和输出设备等五大部件组成。

(2)五大部件的功能分别是：运算器能进行加、减、乘、除等基本运算及附加操作；控制器能自动执行指令；存储器能存储数据和指令，并能区分它们；操作人员可以通过输入/输出设备与主机进行通信。

(3)计算机内部采用二进制来表示指令和数据。每条指令一般具有一个操作码和一个地址码，其中操作码表示运算性质，地址码指出操作数在存储器中的位置。

(4)计算机采用存储程序方式工作。这种工作方式需要事先编制好程序，将程序和数据存入主存储器中，计算机在运行程序时就自动地、连续地从存储器中取出指令、分析指令并执行指令，而无需人工干预。

(5)经典的冯·诺依曼计算机以运算器为中心，其结构如图 1-7 所示。图中实线为数据线，虚线为控制线和反馈线。

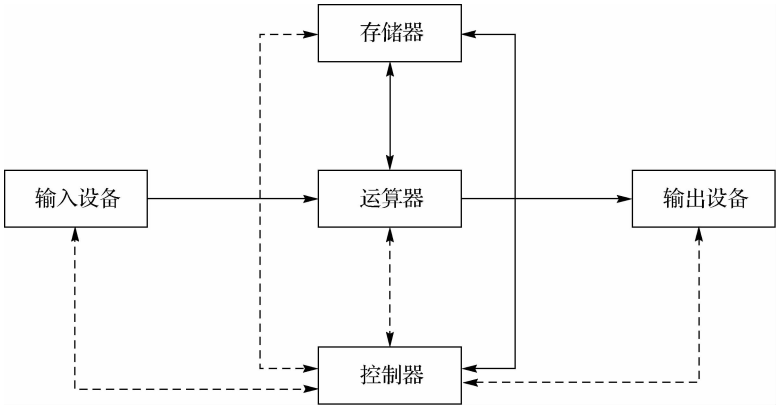


图 1-7　经典的冯・诺依曼计算机结构

虽然随着技术的发展,计算机系统结构有了很大改进,但是原则上变化不大,现代计算机还都是基于冯・诺依曼思想的。计算机的这种工作方式可称为控制流(指令流)驱动方式,而数据信息流则被动地被调用和处理。

1.4.2　现代计算机体系结构

由于经典的冯・诺依曼计算机以运算器为中心,所有的输入和输出操作都要经过运算器,从而导致计算机效率低下,因此现代计算机已转为以存储器为中心,如图 1-8 所示。

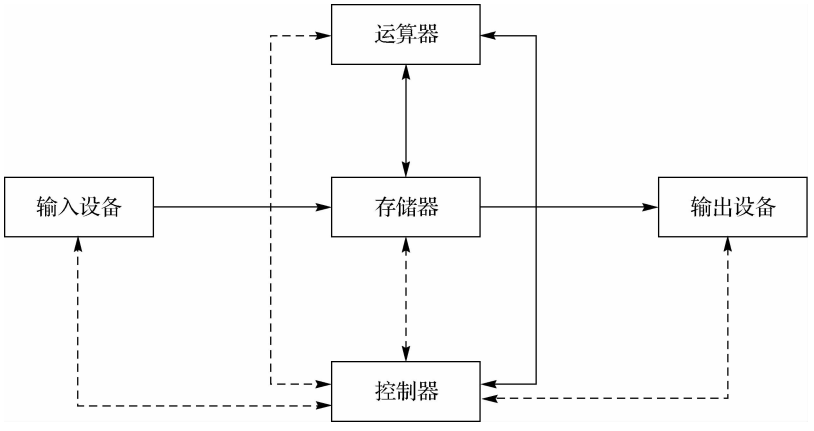


图 1-8　现代计算机结构

随着技术的发展,在现代计算机中运算器与控制器被封装在一起,称为 CPU(central processing unit,中央处理单元),CPU 是计算机硬件的核心。CPU 和主存一起称为主机(main frame),外存和输入/输出设备一起统称为外部设备或外围设备。这样,现代计算机可以认为由 CPU、存储和 I/O 三大子系统组成。

1)CPU 子系统

CPU 有三个组成部分,即 ALU(arithmetic logical unit,算术逻辑单元)、寄存器组和控

制单元，如图 1-9 所示。

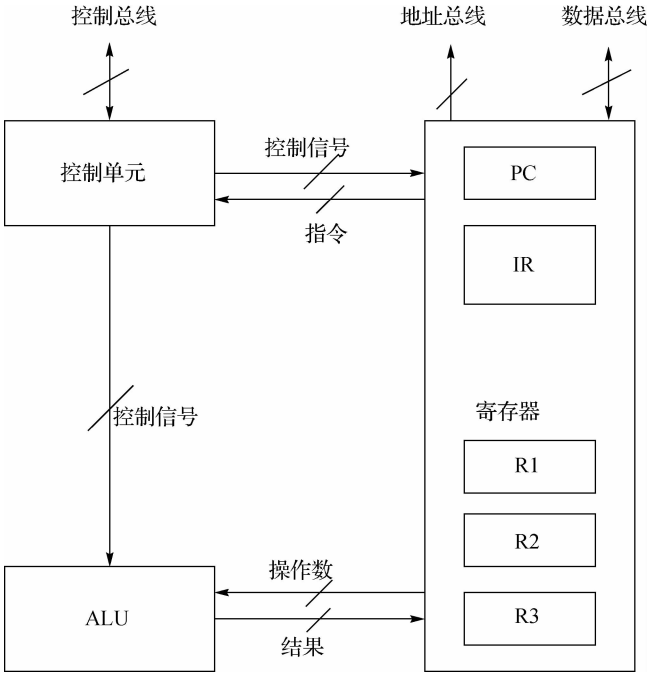


图 1-9　CPU 内部组成

(1)ALU。ALU 是进行算术运算和逻辑运算的部件。算术运算包括加、减、乘、除、加 1、减 1 以及它们的复合运算。逻辑运算包括比较、移位、与、或、非以及异或等操作。ALU 从 CPU 的寄存器部分取得操作数，然后将运算结果再存回到寄存器部分。

(2)寄存器组。寄存器是用于临时存放数据的高速存储设备，CPU 的高速运算离不开多个寄存器。其中一些寄存器可参见图 1-9，主要包括 DR(data register，数据寄存器)、PC(program counter，程序计数器)和 IR(instruction register，指令寄存器)。数据寄存器用于存放参与运算的操作数和运算结果。当代计算机的 CPU 内部设置了大量寄存器来提高运算速度和减少访问存储器的次数。为了简便，图 1-9 中仅给出了 3 个通用寄存器，即寄存器 R1、R2 和 R3，其中两个用来存储输入数据，另一个用来存储输出数据。寄存器的数据均从存储器中取得，其最后结果也存放到存储器中。

程序计数器又称指令计数器，它给出程序中下一条指令在存储器中的单元地址，兼有指令地址寄存器和计数器的功能。当一条指令执行完毕时，PC 作为指令地址寄存器，其内容已变成下一条指令的地址。控制器依据 PC 的内容从存储器取出指令到 IR，同时 PC 将自动加 1，指示下一条指令的地址。若非顺序执行，则只要将 PC 内容作相应改变，就可按新的序列顺序执行指令。

IR 保存当前正在执行的指令代码。一条指令由操作码和地址码两部分组成，其中，OP(operation code，操作码)指出机器操作的类型，如取数、加法、减法等，不同的指令有不同的操作码；地址码用来指示参与操作的数据保存在什么地方，如数据寄存器、主存单元或 I/O 设备等。

(3)控制单元。控制单元(control unit)是计算机的管理机构和指挥中心,它协调计算机的各部件自动工作。在计算机中,程序是由一系列指令构成的,而指令是要求计算机进行基本操作的命令。

控制单元的实质就是解释程序,它每次从存储器中读取一条指令,经过分析译码产生一系列的控制信号,发向各个部件以控制它们的操作。连续不断、有条不紊地继续上述动作就是执行程序。

2)存储子系统

存储器的主要功能是存放数据和程序。

用户对存储器总的要求是容量大、速度快和价格低,但目前这种要求并不总能得到满足。例如,速度快的存储器通常价格高,容量大的存储器通常速度慢等。解决问题的方法是采用多级存储器构成存储层次。目前,存储系统常分为三级,如图 1-10 所示。CPU 能按存储单元地址直接访问主存。增加高速缓冲存储器(cache)的目的是为了提高速度,解决 CPU 与主存之间速度不匹配的矛盾。增加辅存的目的是弥补主存容量的不足。

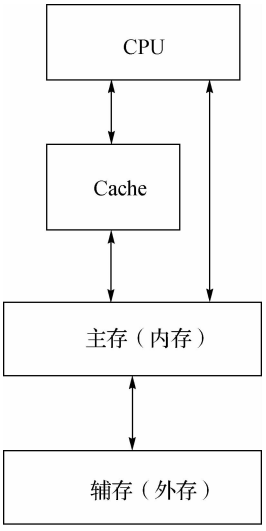


图 1-10　三级存储体系

(1)主存。主存的基本构成是存储元件,它能存储 1 位(bit)二进制信息。若干存储元件按一定拓扑结构排列构成存储单元(cell)或字(word)。数据以字的形式在存储器中传入和传出,一个字可以是 8 位、16 位、32 位,甚至是 64 位。若字是 8 位二进制信息,则称其为一个字节(byte)。目前,用来度量主存容量的单位主要有千字节(KB)、兆字节(MB)和千兆字节(GB)。注意这些术语可能会带来很多误解与混淆,因为这些单位在其他领域中已经被用做 10 的幂,但表示主存容量时,实际的字节数是 2 的幂。即

$$1\ \text{KB}=2^{10}\ \text{B}=1\ 024\ \text{B}$$

$$1\ \text{MB}=2^{20}\ \text{B}=1\ 048\ 576\ \text{B}$$

$$1\ \text{GB}=2^{30}\ \text{B}=1\ 073\ 741\ 824\ \text{B}$$

采用 2 的幂为单位是为了使寻址更加方便。若某存储器容量为 32 KB,则它所需的存储元

件数为 32×1 024×8=262 144。主存中所有存储元件的集合称为存储体，它是主存储器的核心部件。

主存的逻辑结构如图 1-11 所示，它由存储体和外围电路组成。存储体就像一个庞大的仓库，它由许多存储单元组成，每个单元存放一个数据或一条指令。为了区分不同的存储单元，通常将全部单元进行统一编号，此编号称为存储单元的地址码，即单元地址，它用二进制编码表示。不同的存储单元有不同的地址码，单元与单元地址是一一对应的。例如，主存容量是 64 K(2^{16})，字长为一个字节，那么就需要 16 位二进制编码来确定地址，其起始地址通常是 0000000000000000(地址 0)，最后一个地址通常是 1111111111111111(地址 65 535)。一般地，如果某计算机有 N 个字的存储空间，那么就需要有 $\log_2 N$ 位的二进制编码来确定每一个存储单元。注意，每个存储单元只有一个地址，但存储在其中的信息是可以更换的。

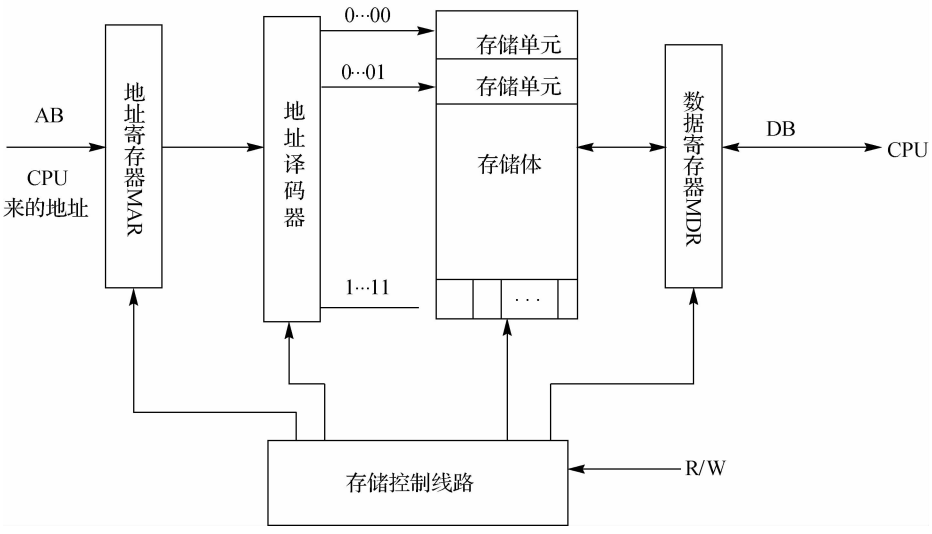


图 1-11　主存储器的组成

在存储单元中存入或取出信息，称为访问存储器，即对存储器进行写入或读出操作。访问存储器时，地址经地址总线送入地址寄存器(MAR)后进行地址译码，以找到相应的存储单元；然后存储控制电路根据读/写命令产生读出或写入时序信号，将存储器中指定单元的数据读出到数据缓冲寄存器(MBR)中，或将数据缓冲寄存器中的数据写入存储器的指定单元。通常读出时，被读出的存储单元的内容不变；写入时，被写入的存储单元中原有内容被破坏而代之以新写入的内容。

(2)高速缓冲存储器。高速缓冲存储器的存取速度比主存快，但比 CPU 内部的寄存器慢。高速缓冲存储器的容量较小，且常被置于 CPU 与主存之间。

高速缓冲存储器在任何时候都只是主存中部分内容的复制。当 CPU 要存取主存中的某个信息时，CPU 首先检查 cache，如果 cache 中有该信息，CPU 就立即存取它；如果 cache 中没有该信息，CPU 就从主存中将包含该信息的一个数据块复制到 cache 中，然后再访问 cache 并读/写该信息。由于计算机中的指令大部分是顺序执行的，很多数据也是顺序存放和处理的(如数组等)，因此 CPU 下次要访问的信息很有可能就是该信息的后续字，这时访

问 cache 即可，从而提高了处理速度。

(3)辅存。辅存用来存放暂时不执行的程序和数据，起支援主存的作用。它不能与 CPU 直接交换信息，只能与主存成批交换信息。因为它设在主机外部，属于外部设备，所以又称为外存储器(简称外存)。辅存的最大特点是存储容量大、可靠性高、价格低，在脱机的情况下也能永久保存信息，但存取速度较慢。

辅存分为磁表面存储器和光存储器。目前使用的磁表面存储器主要有磁盘和磁带，光存储器主要是光盘。

3)I/O 子系统

输入设备(input equipment)的作用是将参加运算的数据和程序送入计算机，并将它们转换成计算机能识别的信息。常见的输入设备有键盘、鼠标、扫描仪、摄像机等，它们大多是电子和机械混合的装置，速度较慢，因此，一般均通过接口与 CPU 或存储器相连接。

输出设备(output equipment)是将计算机处理的结果转化为人或其他设备所能识别或接收的信息形式的装置。常见的输出设备有显示器、打印机、绘图机等。输出设备也大多为机电装置，也需通过接口与 CPU 或存储器连接，如图 1-12 所示。

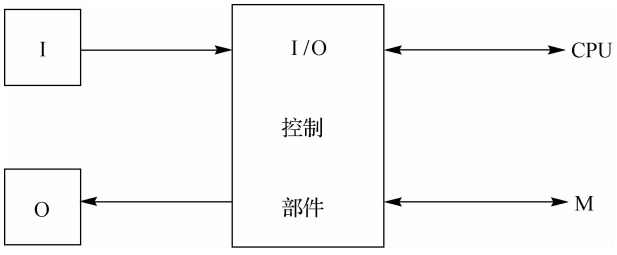


图 1-12　I/O 部件逻辑框图

4)总线

总线(bus)是连接计算机各部分之间进行信息传送的一组公共传输线，它将上述各大部件连接构成一个有机的整体，如图 1-13 所示。总线能分时接收各部件送来的信息，并发送信息到有关部件。采用总线结构后，系统的连接就显得十分清晰、规整，便于设备的扩充和维护。

系统总线通常包括地址总线(address bus)、数据总线(data bus)和控制总线(control bus)。地址总线是单向的，当 CPU 从存储器读取数据或指令，或写数据到存储器时，它必须指明将要访问的存储器单元地址。CPU 输出地址到地址总线上，而存储器从地址总线上读取地址，并且根据它来访问正确的存储单元。每个 I/O 设备，如键盘、显示器或者磁盘等，同样都有一个唯一的地址。当访问某个 I/O 设备时，CPU 将此设备的地址放到地址总线上。每个设备均从总线上读取地址并且判断自己是不是 CPU 将要访问的设备。与其他总线不同，地址总线总是从 CPU 上接收信息，而 CPU 从不读取地址总线。

数据总线主要用于传送各大部件之间的数据信息。当 CPU 从存储器中取数据时，它首先把存储器地址输出到地址总线上，然后存储器输出数据到数据总线上，这样 CPU 就可以从数据总线上读取数据了。当 CPU 向存储器中写数据时，它首先输出地址到地址总线上，

然后输出数据到数据总线上，这样存储器就可以从数据总线上读取数据并将它存储到正确的单元中。对 I/O 设备读/写数据的过程与此类似。

控制总线与以上两种总线都不相同。地址总线由 n 根线构成，n 根线联合传送一个 n 位的地址值。类似地，数据总线的各条线联合起来传输一个单独的多位值。相反地，控制总线是一组独立的控制信号的集合。这些信号用来指示数据是要读入还是写出 CPU，CPU 是要访问存储器还是 I/O 设备，I/O 设备还是存储器已就绪要传送数据等。虽然图 1-13 的控制总线看起来是双向的，但它实际上是一组单向信号的集合。大多数信号是从 CPU 输出到存储器及 I/O 子系统的，只有少数是从这些子系统输出到 CPU 的。后续章节将对此进行详细讨论。

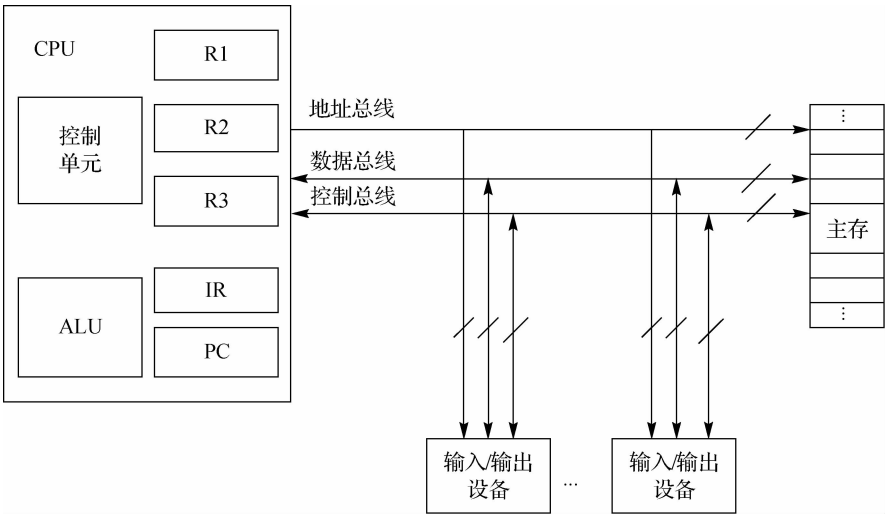


图 1-13　以总线连接的计算机框图

一个系统可以具有总线层次。例如，它可以使用地址总线、数据总线和控制总线来访问存储器和 I/O 控制器；而 I/O 控制器则使用第二级总线来访问所有的 I/O 设备。第二级总线通常称为 I/O 总线(I/O bus)或者局部总线(local bus)，如 PCI 总线等。

1.4.3　计算机的工作过程

计算机的工作过程实质是执行程序的过程，而执行程序的过程就是逐条执行指令的过程，因此，了解指令执行过程是了解计算机工作过程的基础。通过讨论指令执行过程，还将具体地了解计算机各组成部件是如何协调工作的以及它们之间的功能联系。

1)指令执行过程

要执行指令，首先要从存储器取出指令，然后才能执行，因而一般把指令执行过程分为 3 个阶段，即取指令、译码和执行指令，如图 1-14 所示。

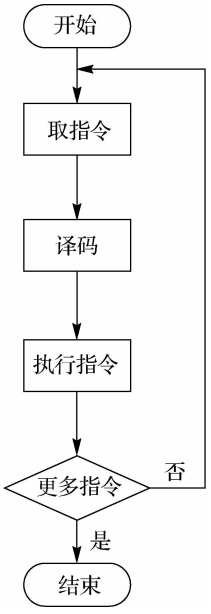


图 1-14　指令执行过程

(1)取指令。在取指令阶段,CPU 根据程序计数器 PC 的内容,将下一条即将要执行的指令从主存复制到指令寄存器中。复制完成后,程序计数器 PC 的内容自动加 1,指向下一条指令。

上述取指令操作与取出指令的内容是无关的,取任何指令都需要这些步骤,因此取指令操作是所有指令的公共操作。

(2)译码。将指令取到 IR 后,由控制部件进行译码,并确定是什么类型的指令。

(3)执行指令。指令译码完成后,控制单元向有关的功能部件发送为执行该指令所需要的一切控制信号。这些信号有些是同一节拍产生的,有些是按序产生的,取决于指令的操作性质。不同的指令有不同的控制信号序列。执行不同的指令,调用的功能部件是不同的。例如,执行转移型指令只需要将新的下一条指令的地址送入 PC 即可,它靠调用控制器本身的部件就可以完成;而执行需要从主存中调入操作数的加法指令时,则需要调用运算器、存储器和控制器。

不论执行什么指令,执行完最后一步操作后都需要回到取指令阶段,等待去取下一条指令。计算机如此周而复始的执行程序中的每条指令,直到整个程序执行完为止。

2)计算机工作过程举例

设有一台计算机,其基本字长为 32 位,通用寄存器有 16 个(需要 4 位二进制寻址),指令操作码为 8 位,存储单元地址为 20 位,为书写方便采用十六进制编码,其部分指令如表 1-10 所示。

表 1-10　指令系统部分指令

指令名称	记 忆 符	OP	第一地址	第二地址	功能说明
取数	LDR	01	R_1	D_2	$R_1 \leftarrow M(D)_2$
存数	STR	02	R_1	D_2	$M(D_2) \leftarrow R_1$
加法	ADD	03	R_1	D_2	$R_1 \leftarrow R_1 + M(D_2)$
乘法	MUL	04	R_1	D_2	$R_1 \leftarrow R_1 \times M(D_2)$
停机	HLT	FF	—	—	机器停止运行

若要在该计算机上求解 $y=ax^2+bx+c$，则首先要确定算法，然后编制程序流程图，接着编写程序，最后在机器上运行。过程如下：

(1)程序流程图。

$$y = ax^2+bx+c$$
$$= (a\times x+b)\times x+c$$

根据变换后的算法，用方框图描绘出计算的步骤如下：

$$\boxed{R_1 \leftarrow a\times x} \rightarrow \boxed{R_1 \leftarrow R_1+b} \rightarrow \boxed{R_1 \leftarrow R_1\times x} \rightarrow \boxed{R_1 \leftarrow R_1+c} \rightarrow y$$

(2)存储单元分配。解题中 a,b,c,x 为已知原始数据，编程时要用到它们，因此编程前必须安排它们的存储单元。

设原始数据分别存放在主存地址为 00407H～0040AH 单元，计算结果保存在 0040BH 单元。

(3)编制程序。设程序的首地址为 00400H，按流程图和表 1-10 指令系统可编制如表 1-11 所示的程序。

表 1-11　计算 y 的程序

地　　址	指令或数据			说　　明
00400	01	1	00407	取数：$R_1 \leftarrow a$
00401	04	1	0040A	乘法：$R_1 \leftarrow a\times x$
00402	03	1	00408	加法：$R_1 \leftarrow ax+b$
00403	04	1	0040A	乘法：$R_1 \leftarrow (ax+b)x$
00404	03	1	00409	加法：$R_1 \leftarrow (ax+b)x+c$
00405	02	1	0040B	存数：0040E$\leftarrow y$
00406	FF			停机
00407	a			原始数据 a
00408	b			原始数据 b
00409	c			原始数据 c
0040A	x			原始数据 d
0040B	y			结果 y

(4)运行程序。编制好程序后,就可在引导程序的控制下,通过输入设备将其输入存储器的指定存储区中了。

程序输入后,引导程序将强迫程序计数器 PC 的内容为程序的首指令地址(对于简单的机器,也可以通过控制台将程序首地址装入 PC,再启动机器运行)。在本例中,PC 被置为 00400H,之后计算机开始执行指令的工作过程。首先从 00400H 单元取指令 01100407H 到 IR,PC 加 1 变为 00401H,IR 的内容经译码识别出是取数指令,在执行指令阶段,将 00407H 单元的数 a 读到 R_1 寄存器;接着控制器又进入取指令阶段,从 00401H 单元中取出指令 0410040AH 到 IR,PC 内容再加 1 变为 00402H,IR 中的指令经译码识别出是乘法指令,于是在执行指令阶段,从 0040AH 单元取出被乘数 x,它与 R_1 中的乘数 a 都送入 ALU 中进行乘法运算,乘积存入 R_1;接着又从 00402H 单元取出新的指令并执行之。如此逐条执行程序中的每条指令,直到从 00406H 单元中取出指令 FFH,执行停机指令,使控制单元不再循环发出节拍信号,机器也就停止了指令执行过程。如果最后 00406H 单元中不是停机指令,那么一般情况下应是一条无条件转移指令,其转移地址是另一程序的首指令地址,之后计算机便从新的地址开始执行新的程序。

习　题　1

(1)计算机的语言分为哪三类?各有何主要特点?

(2)写出下列传送的有效 RTL 语句并给出其相应的硬件实现。所有的寄存器都是 1 位宽。

①IF α=1 THEN copy X to W and copy Z to Y。

②IF α=1 THEN copy X to W;otherwise copy Z to Y。

③IF α=0 THEN copy X to W。

(3)用 RTL 和硬件实现当 α=1 时,关于 8 位寄存器的混洗操作。序列 ABCDEFGH 混洗后得 AEBFCGDH。

(4)在一个组合系统中,用硬件实现下列三个 RTL 语句,要求控制硬件以确保在任意时刻 3 个控制信号 α、β 和 γ 中最多只有一个有效。

①α: X←X+Y。

②β: X←X+Y′+1。

③γ: X←X⊕Y。

(5)简述计算机的发展过程。

(6)计算机系统的主要性能指标有哪些?

(7)冯·诺依曼思想的关键是什么?按其思想计算机由哪几个部件组成?各部件的主要功能是什么?

(8)什么是总线？系统总线根据传送信息的不同可以分为哪三种？

(9)现代计算机的存储系统通常采用哪三级结构？各自的作用是什么？

(10)CPU 由哪几个部分组成？各部分的主要功能是什么？

(11)中央处理器中有哪几个主要寄存器？试说明它们的作用。

(12)结合计算机的组成与结构，详细说明一条机器指令的执行过程。

第 2 章 数据的表示

计算机内部流动的信息可以分为两大类:数据信息和控制信息。数据信息是计算机加工处理的对象,可以分为数值数据和非数值数据。数值数据有确定的值,并在数轴上有对应的点。非数值数据一般用来表示符号或文字,它没有值的含义。本章将讨论数据信息在计算机中的表示,使读者了解计算机中最基本和最常用的一些数据表示方法,认识计算机中的数据表示与人们习惯书写的数的表示形式是不同的。计算机的数据类型能由计算机硬件直接识别,并由计算机指令直接调用。数据表示是影响计算机全局性的问题,也是计算机硬件和软件的接口或界面之一,了解和掌握计算机中的数据表示是了解计算机各主要部件工作的必要基础。

本章首先讨论数制与编码,包括常用的进位计数制及其相互转换,真值和机器数,BCD码,字符与字符串,以及校验码。然后介绍定点数(fixed-point notation),定点数是指小数点的位置固定不变的数,分为无符号数与有符号数。最后介绍浮点数(floating point number),浮点数是指小数点右边的数位可以变化的数据。本章将说明浮点数的格式和性质,同时,还将描述 IEEE 754 浮点数标准,这是所有能处理浮点数的现代 CPU 都遵循的标准。

2.1 数制与编码

在计算机科学概论和数字逻辑等课程中已较详细地讨论了进位计数制及其编码问题,本节仅将有关的基本内容简要归纳如下。

2.1.1 进位计数制及其相互转换

一个数值数据有三个要素:进位计数制、数的正负符号和小数点。

1)常用计数制

按进位的方式计数的数制叫做进位计数制，简称进位制。在日常生活中，人们习惯于用十进制(逢十进一)形式来表示数据。但在计算机内部数据是以二进制(逢二进一)形式表示的，这是因为二进制数只有“0”和“1”两个数字(又称数码)，便于用物理元件或数字电路的两种稳定状态来表示，而且二进制数运算简单，相应的运算线路也十分简单。为了方便用户输入/输出或书写数据，也常用到八进制和十六进制。同一个数用不同的数制表示，这就存在它们之间的相互转换问题。

无论数据采用哪种进位制表示，都涉及两个基本问题，即基数和各数位的权。基数是指该进位制中允许选用的基本数码的个数。例如，十进制数每个数位上允许选用 0，1，2，…，9 共 10 个不同数码中的某一个，因此其基数为 10。而一个数码处于不同的数位上代表着不同的数值，其数值等于该数码乘以一个与其位置相关的常数，这个常数称为权(weight)。

在十进制中，任何一个数$(N)_{10}$可表示为：

$$\begin{aligned}(N)_{10} &= D_m D_{m-1}\cdots D_1 D_0 . D_{-1} D_{-2}\cdots D_{-k}\\ &= D_m \times 10^m + D_{m-1} \times 10^{m-1} + \cdots + D_1 \times 10^1 + D_0 \times 10^0 +\\ &\quad D_{-1} \times 10^{-1} + D_{-2} \times 10^{-2} + \cdots + D_{-k} \times 10^{-k}\\ &= \sum_{i=m}^{-k} D_i \times 10^i \end{aligned} \tag{2-1}$$

其中，$(N)_{10}$的下标 10 表示十进制；m 和 k 为正整数；D_i 为数码 0，1，…，9 中的一个，根据 D_i 在式中所处的位置而赋以一个固定的权值 10^i；式中的 10 为基数。

例如，十进制数 168.75 可写成

$$168.75 = 1\times10^2 + 6\times10^1 + 8\times10^0 + 7\times10^{-1} + 5\times10^{-2}$$

从十进制中得到的关于数制的概念，可以推广到任意的 R 进制数(R 为正整数)。

一个任意 R 进制数可表示为：

$$\begin{aligned}(N)_R &= D_m D_{m-1}\cdots D_1 D_0 . D_{-1} D_{-2}\cdots D_{-k}\\ &= D_m \times R^m + D_{m-1} \times R^{m-1} + \cdots + D_1 \times R^1 + D_0 \times R^0 +\\ &\quad D_{-1} \times R^{-1} + D_{-2} \times R^{-2} + \cdots + D_{-k} \times R^{-k}\\ &= \sum_{i=m}^{-k} D_i \times R^i \end{aligned} \tag{2-2}$$

式中，整数部分有 $m+1$ 位，小数部分有 k 位，R 为基数，R^i 为对应的权，D_i 可以是 0，1，…$R-1$ 共 R 个数字中的任意一个，每个数位计满 R 后就向高位进位，即逢 R 进一。

基数 $R=2$ 时，为二进制数，权为 2^i，D_i 为 0，1 两个数字中的一个，逢二进位。

基数 $R=8$ 时，为八进制数，权为 8^i，D_i 为 0，1，…，7 八个数字中的一个，逢八进位。

基数 $R=16$ 时，为十六进制数，权为 16^i，D_i 为 0，1，…，8，9，A，B，C，D，E，F 十六个数字中的一个(A，B，C，D，E，F 与十进制数字的对应关系分别为 10，11，12，13，14，15)，逢十六进位。

十进制、二进制、八进制和十六进制之间的对照如表 2-1 所示。

表 2-1　十进制、二进制、八进制和十六进制之间的对照

十 进 制	二 进 制	八 进 制	十六进制
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

2)数制之间的转换

(1)R 进制数转化为十进制数。把任意 R 进制数按照式(2-2)写成按权展开式后再求和,就可得到该 R 进制数对应的十进制数。

【例 2-1】 $(1101.0101)_2=1\times2^3+1\times2^2+0\times2^1+1\times2^0+0\times2^{-1}+1\times2^{-2}+0\times2^{-3}+1\times2^{-4}$

$=(8+4+0+1+0+0.25+0+0.0625)_{10}$

$=(13.3125)_{10}$

【例 2-2】 $(731.6)_8=7\times8^2+3\times8^1+1\times8^0+6\times8^{-1}$

$=(448+24+1+0.75)_{10}$

$=(473.75)_{10}$

【例 2-3】 $(2D.5)_{16}=2\times16^1+13\times16^0+5\times16^{-1}$

$=(32+13+0.3125)_{10}$

$=(45.3125)_{10}$

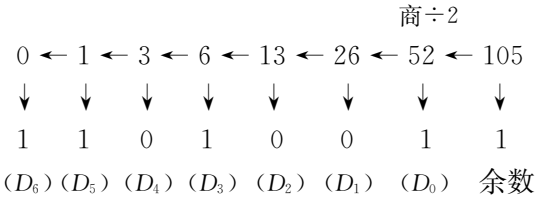
(2)十进制数转化为 R 进制数。十进制数的整数部分与小数部分的转换方法不同,应分别转换,然后将两部分合并才能得到结果。

①整数部分转换方法——除基取余法。先将十进制数的整数部分除以基数 R,取余数,该余数为 R 进制数最低位,即第 0 位的数码 D_0;再以求得的商除以基数 R,取余数,即为第 1 位的数码 D_1;以此类推,直到求得商为 0 为止。将所得的余数按序排列所得出的数,就是 R 进制数的整数部分。

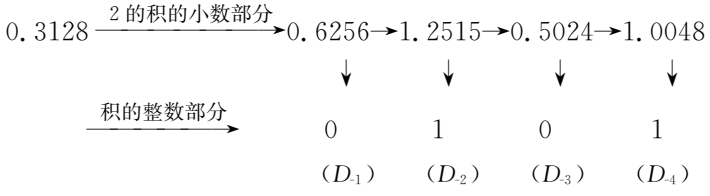
②小数部分转换方法——乘基取整法。先将十进制数的小数部分乘以基数 R，取其积的整数部分，该整数即为 R 进制小数部分中小数点后第一位的数码 D_{-1}；再以基数 R 乘以所得积的小数部分，新积的整数即为第二位的数码 D_{-2}；以此类推，直到乘积的小数部分为 0，或结果已满足所需精度要求为止。

【例 2-4】 将 $(105.3128)_{10}$ 转换为二进制数（小数部分要求 4 位有效值）。

解：整数部分：



小数部分：



所以 $(105.3128)_{10} \approx (1101001.0101)_2$

（3）二进制数、八进制数和十六进制数之间的转换。八进制数和十六进制数是从二进制数演变而来的，由 3 位二进制组成 1 位八进制数，4 位二进制数组成 1 位十六进制数。对于一个兼有整数和小数部分的数，以小数点为界，对小数点前后的数分别分组进行处理，不足的位数用 0 补充，对整数部分将 0 补在数的左侧，对小数部分将 0 补在数的右侧。这样数值不会发生差错。

若从二进制数转换为八进制数，则以 3 位为一组（以下划线表示）。

【例 2-5】 $(1011.1101)_2 = (\underline{001}\ \underline{011}.\underline{110}\ \underline{100})_2$

$= (13.64)_8$

若从二进制数转换为十六进制数，则以 4 位为一组（以下划线表示）。

【例 2-6】 $(101011.110110)_2 = (\underline{0010}\ \underline{1011}.\underline{1101}\ \underline{1000})_2$

$= (2B.D8)_{16}$

从八进制数或十六进制数转换为二进制数，只要顺序将每一位数写成 3 位或 4 位二进制数即可。

【例 2-7】 $(15.64)_8 = (\underline{001}\ \underline{101}.\underline{110}\ \underline{100})_2$

$= (1101.1101)_2$

【例 2-8】 $(7D.E6)_{16} = (\underline{0111}\ \underline{1101}.\underline{1110}\ \underline{0110})_2$

$= (1111101.1110011)_2$

八进制数与十六进制数之间的转换，可用二进制数作为中间媒介进行转换。

可以看出，八进制数或十六进制数的形式比二进制数简短，而且又很容易转换成二进制数，因此，计算机工作者经常使用八进制数或十六进制数。为清晰起见，常在数字后面加字母 B 表示二进制数（binary），O 表示八进制数（octal），H 表示十六进制数（hexadecimal），D 或不加字母表示十进制数（decimal）。

2.1.2 真值和机器数

人们通常在数据的绝对值前面加上“+”或“-”来表示数的正或负，然而在计算机中符号必须数码化。通常采用的方法是在数据的前面增设一位符号位，0 表示“+”，1 表示“-”。这种在计算机中使用的、连同数字和符号一起数码化的数称为机器数。而按一般习惯书写的形式，即正、负号加绝对值表示的数称为机器数的真值。

【例 2-9】 真值为+1011 的一种机器数形式为 01011；

真值为-1011 的一种机器数形式为 11011。

计算机中常用的机器数表示方法有 3 种，即原码、补码和反码，其中原码和补码最常用。

1)原码表示

原码表示也称为符号—幅值表示，是机器数中最简单、最直观的一种表示方法。它约定数码序列中的最高位为符号位，符号位为 0 表示该数为正，为 1 表示该数为负；其余有效数值部分则用二进制的绝对值表示。

【例 2-10】 若 X=+0.1011，则$[X]_{原}=0.1011$；

若 X=-0.1011，则$[X]_{原}=1.1011$；

若 X=+1101，则$[X]_{原}=01101$；

若 X=-1101，则$[X]_{原}=11101$。

后面讨论定点表示时将要指出，在计算机中常将数据规范化为纯小数(定点小数)或纯整数(定点整数)。下面分别给出定点小数与定点整数的原码表示，可以作为推导有关运算法则的基础。

若小数的原码序列为 $X_0.X_1X_2\cdots X_n$，其中 X_0 为符号位，则

$$[X]_{原}=\begin{cases}X & 1>X\geqslant 0\\ 1-X=1+|X| & 0\geqslant X>-1\end{cases} \tag{2-3}$$

其中，X 表示真值，$[X]_{原}$为原码表示的机器数。当 X 为正时，$[X]_{原}$与 X 相同；当 X 为负时，$[X]_{原}=1+|X|$，即符号位为 1，再加上小数部分的绝对值，这与原码表示法的约定相同。

若整数的原码序列为 $X_nX_{n-1}X_{n-2}\cdots X_0$，其中 X_n 为符号位，则

$$[X]_{原}=\begin{cases}X & 0\leqslant X<2^n\\ 2^n-X=2^n+|X| & -2^n<X\leqslant 0\end{cases} \tag{2-4}$$

与小数相似，当 X 为正时，$[X]_{原}$与 X 相同；当 X 为负时，$[X]_{原}=2^n+|X|$，其中 2^n是符号位的权值，加 2^n相当于令符号位为 1。注意，在式(2-3)与式(2-4)中，有效数位是 n 位，连同符号位是 $n+1$ 位。

分析该定义式，可以得出原码的性质。

(1)零的表示有“+0”和“-0”之分，以定点小数为例，表示如下：

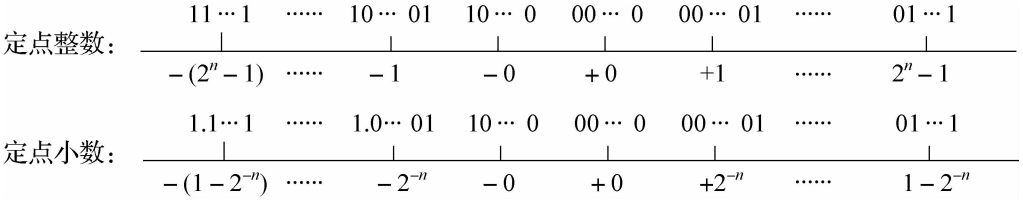
$$[+0]_{原}=0.0\cdots 0$$

$$[-0]_{原}=1.0\cdots 0$$

(2)符号位不是数值的一部分，它们是人为约定的，所以符号位在运算过程中需要单独处理，不能作为数值的一部分直接参与运算。

(3)原码表示的纯小数范围为$-1<X<1$,即$|X|<1$。原码表示的纯整数,其范围为$-2^n<X<2^n$,即表示范围限制在$|X|<2^n$。

(4)可用数轴表示出原码的表示范围和可能的代数组合,数轴上方表示的是原码的代数组合,下方注明原码对应的真值。



原码表示法的优点是比较直观,缺点是加减运算复杂。

2)补码表示

设置补码表示法有两个目的:一是使符号位也作为数值的一部分直接参与运算,简化加减运算方法,节省运算时间;二是使减法运算转化为加法运算,从而进一步简化计算机中运算器的线路设计。

(1)模数的概念。先看两个十进制数的运算。

$$79-38=41$$

$$79+62=141$$

如果使用两位数的运算器做(79+62)的运算,那么结果中多余的 100 因为超出了运算器两位数的范围而将自动丢弃,这样在做(79−38)的减法时用(79+62)的加法同样得到正确的结果。类似的例子很多,在数学上可以用同余式表示为

$$79-38=79+(100-38)\pmod{100}$$

这里的 100 称为模,用 M 或 mod 表示。模是指一个计算系统的计算范围,即产生溢出的量。两位十进制数的计算范围为 00～99,溢出量为 100,模就是 100,上述运算也称为模运算,可以写为

$$79+(-38)=79+62\pmod{100}$$

可进一步写为

$$-38=62\pmod{100}$$

即−38 的补码(模为 100)是 62。一个负数(如−38)用其补码(如+62)代替,同样可得到正确的结果。

在计算机中,机器能表示的数据位数是固定的,因此它的运算也是一种有模运算。如果数据有 $n+1$ 位(包括一位符号位),那么它的模为 2^{n+1}。模在机器中是表示不出来的,若运算结果超出了能表示的数据范围,则只保留它的小于模的低 n 位数码,超出 n 位的高位部分就自动丢弃了。

由上述分析可得:当负数用其补码表示时,可将减法转化为加法,在计算机中实现起来比较方便。

(2)补码的定义。确定模之后,将某数 X 对该模的补数称为补码,定义如下:

$$[X]_{补}=M+X\pmod{M}\tag{2-5}$$

若 $X>0$,则模 M 作为正常的溢出量可以舍去。因而正数的补码就是其本身,形式上与原码相同。若 $X<0$,则$[X]_{补}=M+X=M-|X|$。因而负数的补码等于模 M 减去该数的绝

对值。式(2-5)是补码的统一定义式,由此可导出定点数的补码定义。

若定点小数的补码序列为 $X_0.X_1X_2\cdots X_n$,其中 X_0 为符号位,则

$$[X]_{补}=2+X \pmod 2 = \begin{cases} X & 0\leqslant X<1 \\ 2+X & -1\leqslant X\leqslant 0 \end{cases} \tag{2-6}$$

式中 X 为真值,$[X]_{补}$ 是采取补码表示的机器数。

【例 2-11】 若 $X=+0.1011$,则$[X]_{补}=0.1011$;

若 $X=-0.1011$,则$[X]_{补}=2-0.1011=1.0101$。

若定点整数的补码序列为 $X_nX_{n-1}X_{n-2}\cdots X_0$,其中 X_n 为符号位,则

$$[X]_{补}=2^{n+1}+X \pmod{2^{n+1}} = \begin{cases} X & 0\leqslant X<2^n \\ 2^{n+1}+X & -2^n\leqslant X\leqslant 0 \end{cases} \tag{2-7}$$

【例 2-12】 若 $X=+1011$,则$[X]_{补}=01011$;

若 $X=-1011$,则$[X]_{补}=2^{n+1}+X=10000-1011=10101$

式(2-5)是补码的统一定义式,而式(2-6)与式(2-7)则给出了不同数域内真值与补码的对应关系,它们可作为推导有关补码运算法则的依据。

在一些书中,将这种以 2 为基数的补码简称为“2 的补码”。注意,它的含义并非指数值 2 的补码。

(3)由真值、原码转换为补码。根据补码定义,可按式(2-6)和式(2-7)由真值求得补码表示,但比较原码与补码在形式上的差异,可找到更简便的实用转换规律,即由原码求补码。而原码与真值的区别仅在于符号数码化。

①正数的补码表示与正数的原码表示相同。由负数原码表示转换为负数补码表示时,符号位保持不变,其他各位先取反,然后再在末位加 1。这条规律可简称为“取反加 1”。

②正数的补码表示与正数的原码表示相同。由负数原码表示转换为负数补码表示时,符号位保持不变,其他各位自低位向高位,第一个 1 及其以前的各位 0 保持不变,以后的各位按位取反。

【例 2-13】 若$[X]_{原}=01010$,则$[X]_{补}=01010$;

若$[X]_{原}=1.1011$,则$[X]_{补}=1.0101$;

若$[X]_{原}=11010$,则$[X]_{补}=10110$。

在计算机中常采用第一种方法,即利用寄存器的反相输出在加法器中加 1 实现求补。在手算或低速串行求补时,可采用第二种方法。

(4)由补码表示求其真值与原码。计算机的运算结果常呈补码表示形态,可应用上述两种方法将其转换为真值形式或比较直观的原码形式。

【例 2-14】 若$[X]_{补}=01010$,则$[X]_{原}=01010$,$X=01010$;

若$[X]_{补}=10101$,则$[X]_{原}=11011$,$X=-01011$;

若$[X]_{补}=1.0110$,则$[X]_{原}=1.1010$, $X=-0.1010$。

(5)补码的性质。

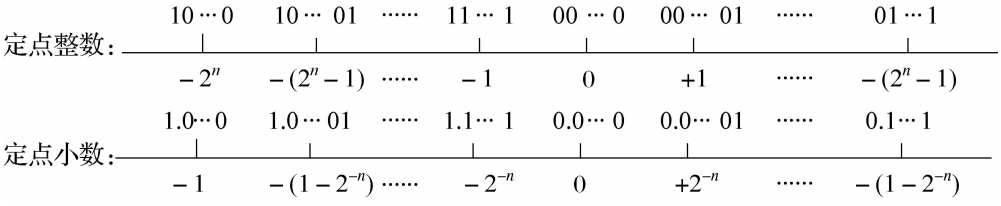
①在补码表示中,真值±0 只有唯一的形式,即

$$[+0]_{补}=[-0]_{补}=\begin{cases}0.00\cdots0 & (小数)\\ 000\cdots0 & (整数)\end{cases}$$

②在补码表示中,最高位 X_0(符号位)表示数的正负,在形式上与原码相同,即 0 正 1 负。但补码的符号位是数值的一部分,由补码定义式计算而得,例如,负小数补码中 X_0 为 1,这个 1 就是真值 X 加模 2 后产生的。

③负数补码的表示范围比原码稍宽,多一种数码组合。对于定点小数,负数补码的表示范围可至−1,代码组合为 1.0…0。对于定点整数,负数补码表示范围可至-2^n,代码组合为 10…0。

④将负数 X 的真值与其补码$[X]_{补}$作一映射图,可以进一步看出:负数补码表示的实质是将负数映射到正数域,因而可达到化减为加、简化运算的目的。



以定点整数为例,根据补码定义式可知:在引入模 2^{n+1} 后,用$[X]_{补}=2^{n+1}+X$ 表示负数 X,相当于将负数 X 向数轴正向平移 2^{n+1},使负数 X 被映射到正域中。于是减一个正数,即加一个负数被转化为加另一个正数,这个正数就是负数补码的映射值。$[X]_{补}$中的符号位 X_0 是映射值中的最高数位,因此在补码运算中符号位像数码一样直接参与运算。

3)反码表示

如果只对原码的尾数逐位取反,就得到了另一种机器数表示形式,即反码,它也能达到化减为加的目的。由于它与补码相比末位少加了一个 1,因此不难由补码定义推得反码定义。

$$[X]_{反}=\begin{cases}X & 0\leqslant X<1\\ (2-2^{-n})+X & -1<X\leqslant 0\end{cases}\tag{2-8}$$

若定点小数 X 的反码形式为 $X_0.X_1X_2\cdots X_n$,其中 X_0 为符号位,见下例。

【例 2-15】 若 $X=+0.1101$,则$[X]_{反}=0.1101$;

若 $X=-0.1001$,则$[X]_{补}=(2-2^{-4})+(-0.1011)=1.0110$。

$$[X]_{反}=\begin{cases}X & 0\leqslant X<2^n\\ (2^{n+1}-1)+X & -2^n<X\leqslant 0\end{cases}\tag{2-9}$$

若定点整数 X 的反码形式为 $X_nX_{n-1}X_{n-2}\cdots X_0$,其中 X_n 为符号位,见下例。

【例 2-16】 若 $X=+1101$,则$[X]_{反}=01101$。

若 $X=-1101$,则$[X]_{反}=(2^5-1)+(-1101)=11111-1101=10010$。

对于 0,反码有“+0”和“−0”之分,即

$[+0]_{反}=0.0\cdots0$,$[-0]_{反}=1.1\cdots1$(小数);

$[+0]_{反}=00\cdots0$,$[-0]_{反}=11\cdots1$(整数)。

因此，$n+1$ 位反码(包括一位符号位)也只能表示 $2^{n+1}-1$ 个数，与原码相同。

从上述讨论可以得出以下结论。

(1)正数的原码、补码和反码有相同的形式。

(2)负数的原码比较直观，即符号位为“1”，其数值部分与真值绝对值相同。

(3)负数的补码中，符号位为“1”，数值部分为原码的数值各位取反后末位加 1 获得。

(4)负数的反码中，符号位为“1”，数值部分为原码的数值各位取反后获得。

综上所述，求与真值对应的机器数，可不按它们的数学定义去求，只要掌握上面的规律就可以方便地得出其原码、补码和反码。

【例 2-17】 已知 $X=+1011011$，求 X 的原码、补码和反码。

解：$[X]_{原}=[X]_{补}=[X]_{反}=01011011$。

【例 2-18】 已知 $X=-1011011$，求 X 的原码、补码和反码。

解：$[X]_{原}=11011011$，$[X]_{补}=10100101$，$[X]_{反}=10100100$。

2.1.3 BCD 码

用二进制位来表示二进制数的存储效率最高，因为每一比特都表示一个唯一有效的值。而在某些应用中，用二进制表示数据并不适合，例如，考虑一个数字钟，它的输出必须总是表示为十进制数，但它的内部元件可以采用二进制数计时，在输出时再将二进制数转换成十进制数。

在数字钟的应用中，一个更好的存储格式是一序列十进制数，而十进制数的每一位都用等价的二进制数表示。尽管这种格式的存储效率比标准的二进制数表示的存储效率要低，但它不需要进行二进制数与十进制数的转换。对于应用来说，消除数制转换所提高的性能足以弥补采用这种存储格式所降低的性能。

用来表示十进制数的最常用格式是 BCD(binary coded decimal)码。

BCD 码用 4 位等值的二进制数表示一个十进制数字。例如，0000 表示 0，1001 表示 9。由于大于 9 的 4 位二进制数不表示任何一个十进制数字，因此在 BCD 码中不使用它们，即不用 1010～1111 的数值。浪费这些数值是使用 BCD 码所要付出的代价。

在 BCD 码中，n 位十进制数用 n 组 4 位二进制数保存。例如，27 被存为 00100111(在二进制中，它被存为 00011011)。这种格式能够表示任何数值。

BCD 码是一种带符号表示法，它的值可以为正也可以为负或 0。类似于符号—幅值表示法，它有两个部分，其中第 1 位用于表示符号，幅值部分保存数的绝对值。

例如，在 BCD 码中，+273 表示为 0001001110011；而−273 表示为 1001001110011。

2.1.4 字符与字符串

字符和字符串是计算机中使用最多的非数值数据，它是人和计算机之间相互交流的桥梁。例如，在大多数计算机系统中，操作人员通过键盘上的字符键向计算机输入各种操作命令和原始数据；计算机则把处理的结果以字符的形式输出到显示终端或打印机上，供操作者使用。

由于计算机内部只能识别和处理二进制代码，所以这些字符必须按照一定的规则用一

组二进制编码来表示。字符编码方式有很多种，现在使用最广泛的是 ASCII 码。

1）ASCII 码

常见的 ASCII（American standard code for information interchange）码用 7 位二进制数表示一个字符，它包括 10 个十进制数字（0～9），52 个大、小写英文字母（A～Z，a～z），34 个专用符号和 32 个控制符号，共计 128 个字符。如表 2-2 所示。

在计算机中，通常一个字节存放一个字符。对于 ASCII 码来说，一个字节右边的 7 位表示不同的字符代码，最左边 1 位既可以用做奇偶校验位，用来检查错误，也可以用于西文字符和汉字的区分标识。

表 2-2　ASCII 字符编码

位数				W_7	0	0	0	0	1	1	1	1
				W_6	0	0	1	1	0	0	1	1
位数				W_5	0	1	0	1	0	1	0	1
W_4	W_3	W_2	W_1	列 行	0	1	2	3	4	5	6	7
0	0	0	0	0	空白（NUL）	转义（DLE）	SP	0	@	P	、	p
0	0	0	1	1	序始（SOH）	机控$_1$（DC$_1$）	!	1	A	Q	a	q
0	0	1	0	2	文始（STX）	机控$_2$（DC$_2$）	"	2	B	R	b	r
0	0	1	1	3	文终（ETX）	机控$_3$（DC$_3$）	#	3	C	S	c	s
0	1	0	0	4	送毕（EOT）	机控$_4$（DC$_4$）	$	4	D	T	d	t
0	1	0	1	5	询问（ENQ）	否认（NAK）	%	5	E	U	e	u
0	1	1	0	6	承认（ACK）	同步（SYN）	&	6	F	V	f	v
0	1	1	1	7	告警（BEL）	组终（ETB）	'	7	G	W	g	w
0	0	0	0	8	退格（BS）	作废（CAN）	(	8	H	X	h	x
1	0	0	1	9	横表（HT）	载终（EM）	)	9	I	Y	i	y
1	0	1	0	10	换行（LF）	取代（SUB）	*	:	J	Z	j	z
1	0	1	1	11	纵表（VT）	扩展（ESC）	+	;	K	[	k	{
1	1	0	0	12	换页（FF）	卷隙（FS）	,	<	L	\	l	|
1	1	0	1	13	回车（CR）	群隙（GS）	-	=	M	]	m	}
1	1	1	0	14	移出（SO）	录隙（RS）	.	>	N	↑	n	ESC 换码
1	1	1	1	15	移入（SI）	元隙（US）	/	?	O	←	o	DEL 抹掉

由表 2-2 可见，数字和英文字母都是按顺序排列的，只要知道其中一个的二进制代码，不用查表就可以推导出其他数字或字母的二进制代码。另外，若将 ASCII 码中 10 个数字的二进制代码去掉最高三位 011，则正好与它们的二进制值相同，这不仅使十进制数字进入计算机后易于压缩成 4 位代码，而且也便于进一步处理机内信息。

2）字符串

字符串是指一串连续的字符。通常，它们在存储器中占用一片连续的空间，每个字节存放一个字符代码，字符串的所有元素（字符）在物理上是邻接的，这种字符串的存储方法称为向量法。例如，字符串“IF X>0 THEN READ(C)”，在字长为 32 位的存储器中的存储格式如图 2-1(a)所示，图中每一个主存单元可存放 4 个字符，整个字符串需要 5 个主存单元。在

每个字节中实际存放的是相应字符的 ASCII 码，如图 2-1(b)所示。

字符串的向量存放法是最简单、最节省存储空间的方法。但是，当字符串需要进行删除和插入操作时，删除或插入子串后的字符串要全部重新分配存储空间，因此花费的时间较多。为了克服向量存放法的缺点，另一种字符串的存储方法，即串表法应运而生。在这种存储方法中，字符串的每个字符代码后有一个链接字，用以指出下一个字符的存储单元地址。串表法不要求串中的各个字符在物理上相邻，一般地，串中各字符可以安排在存储器中的任意位置。在对字符串进行删除和插入操作时，只需修改相应字符代码后面的链接字即可，所以非常方便。但是，由于链接字占据了存储单元的大部分空间，因此使主存的有效利用率下降。例如，一个存储单元有 32 位，仅存放一个字符代码，而链接字占用了 24 位，这时，存放串信息的主存有效利用率只有 25%，这是串表法的最大缺点。

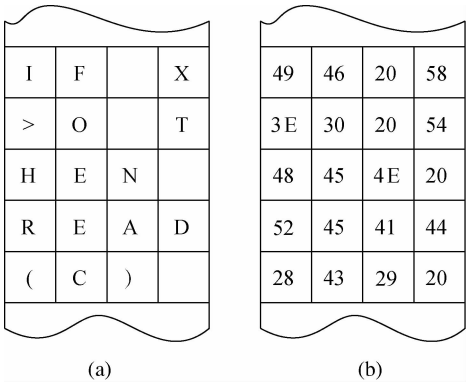


图 2-1　字符串的向量存储方式

2.1.5　校验码

计算机系统中的数据在读/写和传送的过程中都可能产生错误。为减少和避免这类错误，一方面是精心设计各种电路，提高计算机硬件本身的可靠性，另一方面是在数据编码上找出路，即采用带有某种特征能力的编码方法，利用少量的附加电路，使之能发现某些错误，甚至能确定错误的位置，进而实现自动纠错的能力。

数据校验码是一种常用的带有发现某些错误或自动纠错能力的数据编码方法。它的实现原理是加进一些冗余码，使合法数据编码出现某些错误时成为非法编码。这样，就可以通过检测编码的合法性来达到发现错误的目的。合理地安排非法编码数量和编码规则，可以提高发现错误的能力，甚至达到自动纠正错误的目的。这里用到一个码距的概念，码距是指任意两个合法码之间至少有几个二进制位不相同，仅有一位不同时，称其码距为 1。例如，用 4 位二进制数表示 16 种状态，则 16 种编码都用到了，此时码距为 1，也就是说，任何一个状态的四位码中的一位或几位出错，则变成另一个合法码，此时无查错能力。若用 4 个二进制位表示 8 个状态，则可以只用其中的 8 种编码，而把剩余 8 种编码作为非法编码，此时可使合法码的码距为 2。一般地，合理地增大码距能提高发现错误的能力，但编码所使用的二进制位数变多，增加了数据存储的容量或数据传送的数量。在确定与使用数据校验码时，通常要考虑在不过多增加硬件开销的情况下，尽可能发现更多的错误，甚至能自动改正某些最常

出现的错误。常用的数据校验有奇偶校验、海明校验和循环冗余校验。

1)奇偶校验

奇偶校验是一种最简单、应用最广泛的校验方法，常用于主存的读/写检查。其思想是根据代码字的奇偶性质进行编码与校验，有以下两种可供选用的校验规律：

奇校验——使整个校验码(有效信息位和校验值)中“1”的个数为奇数；

偶校验——使整个校验码中“1”的个数为偶数。

有效信息本身不一定满足约定的奇偶性质，但增设校验位后可使整个校验码符合约定的奇偶性质。如果两个有效信息代码之间有一位不同(至少有一位不同)，那么它们的校验位也应不同，因此奇偶校验码的码距为 2。该方法只能发现奇数个错，但不能判断是哪位出错，因此没有纠错能力。

在主存中，通常以字节(字)为单位进行奇偶校验。

【例 2-19】

待编有效信息：10110011

奇校验码(配校验位后)：101100110

偶校验码(配校验位后)：101100111

【例 2-20】

待编有效信息：10100101

奇校验码(配校验位后)：101001011

偶校验码(配校验位后)：101001010

为了快速地进行奇(偶)编码写入以及读出后的奇(偶)校验，多采用并行奇偶统计方法，其逻辑电路可由若干异或门构成，如图 2-2 所示。这种塔形结构同时给出了“奇形成”、“奇校验出错”、“偶形成”、“偶校验出错”。若机器选用偶校验方式，则可取消“奇形成”与“奇校验出错”两个信号。

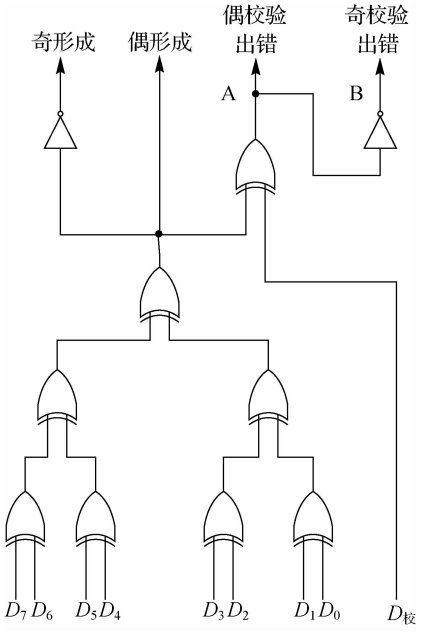


图 2-2　奇偶校验的形成及校验

2)海明校验

海明校验是由 Richard Hamming 于 1950 年提出的一种校验方法，它实际上是一种多重奇偶校验，即将代码按照一定规律组织为若干小组，分组进行奇偶校验，各组的检验信息组成一个指误字，不仅能检测是否出错，而且在只有一位出错的情况下可指出是哪一位错，并且将该位自动地变反以纠正。下面通过一个实例说明其编码方法、查错与纠错功能。

【例 2-21】 待编信息 $A_1A_2A_3A_4$，进行海明校验编码后，要求能发现并纠正一位错。

(1)分成几组？增设多少校验位？设待编有效信息为 k 位，分成 r 组，每组增设一个校验位，共需增设 r 位校验位，组成一个 n 位的海明校验码。校验时每组产生一位校验信息，组成一个 r 位的指误字，可指出 2^r 种状态，其中全 0 表示无错，余下的组合可分别指明 2^r-1 位中的某一位错误。因此，从信息量的角度，如果要求海明校验码能发现并纠正一位错，那么应满足下述关系：

$$n=k+r \leqslant 2^r-1$$

若 $k=4$，则 $r\geqslant 3$，组成 7 位海明码。

(2)分组方法。待编有效信息 $A_1A_2A_3A_4$，增设校验位 $P_1P_2P_3$，分为三组校验，可产生三位的指误字 $G_3G_2G_1$。为了使指误字指明出错位，要求两者之间存在唯一的对应关系。本例采取一种最简单的对应关系，即让指误字代码与出错位序号相同。例如，将 A_1 安排在第 3 位，让它参加第 1 组与第 2 组的校验。若第 3 位 A_1 出错，则 A_1 未参加的第 3 组检错信息为 0，而 A_1 参加了的第 1 组和第 2 组检错信息均为 1，于是产生指误字 $G_3G_2G_1=011$，说明是第 3 位出错。按照这一分组原则，7 位海明校验码的序号与分组关系如表 2-3 所示，每一位参加哪些组，以“√”标注。

表 2-3 ($k=4,r=3$)的海明编码

序号 / 含义 / 分组	1	2	3	4	5	6	7	指误字	正确	出错位						
	P_1	P_2	A_1	P_3	A_2	A_3	A_4			1	2	3	4	5	6	7
第 3 组				√	√	√	√	G_3	0	0	0	0	1	1	1	1
第 2 组		√	√			√	√	G_2	0	0	1	1	0	0	1	1
第 1 组	√		√		√		√	G_1	0	1	0	1	0	1	0	1

每个校验位只参加一组奇偶校验，将 $P_1P_2P_3$ 分别安排在第 1、2、4 位，并分别参与相应的第 1、2、3 组的校验。这是因为每个校验位只是为某一组奇偶校验设置的，与其他组无关，所以只参与一组校验。换句话说，每组只有一位校验位。由于要求指误字能反映出错位的序号，所以将 P_1 安排在第 1 位，只参加第 1 组校验，一旦 P_1 出错，指误字为 001，表明第 1 位出错。同理，将 P_2 安排为第 2 位，只参加第 2 组；将 P_3 安排为第 4 位，只参加第 3 组。于是，三组的奇偶校验表达式如下：

$$G_3=P_3 \oplus A_2 \oplus A_3 \oplus A_4$$
$$G_2=P_2 \oplus A_1 \oplus A_3 \oplus A_4$$
$$G_1=P_1 \oplus A_1 \oplus A_2 \oplus A_4$$

有效信息 $A_1A_2A_3A_4$ 分别参加两组以上的校验，它们依次占据剩下的第 3、5、6、7 位，并

分别参加与此相应的组别，即(1,2)，(1,3)，(2,3)，(1,2,3)。由于所有各位参加的组别并不完全重复，一旦某位出错，指误字将与出错位序号存在唯一的对应关系。

(3)编码。假定每组都采取偶校验，即让 1 的个数为偶数。现欲向外存写入有效信息 1010，将它们分别填入第 3、5、6、7 位，再分组进行奇偶统计，按偶校验要求填入校验位 $P_1P_2P_3$。如果参加第 1 组偶校验的有 $P_1A_1A_2A_4$，P_1 应为 1 才能使该组 1 的个数为偶数，同理 $P_2=0$，$P_3=1$。最后得到 7 位海明校验码为 1011010。

(4)查错与纠错(译码)。假设从外存读出的 7 位海明码为 1011010，按表 2-3 分组进行奇偶检测。由于两个小组都满足偶校验要求，即 $G_3G_2G_1=000$，表明收到的编码是正确的，因此可从中提取有效信息 $A_1A_2A_3A_4$。

如果从外存读出的代码为 1011110，同样按表 2-3 分组进行奇偶检测，得到两组的检错信息，形成指误字 $G_3G_2G_1=101$，表明第 5 位(即 A_2)出错。将第 5 位变反即可纠正错码。

由于海明校验的实质是分组奇偶校验，因此它的编码与查错逻辑与图 2-2 相似。但海明校验具有自动纠错能力，可将由 3 位检错信息构成的指误字 $G_3G_2G_1$ 进行译码，除全 0 外，其余 7 种译码输出分别控制 7 路异或门，控制对应位读出信息是否需要变反纠正。如【例 2-21】，第 5 位异或门的一路输入为 A_2，另一路输入由指误字译码器提供 1，$1\oplus A_2=\overline{A_2}$，变反纠正后输出。限于篇幅，在此不再画出逻辑电路，读者可按描述自行设计。在($k=4$，$r=3$)的海明码中，两组合法编码之间至少有 1 位有效信息不同，由于有效信息位 A_i 至少参加两组校验，相应的两组校验位也将随之不同，因此这种海明码的码距为 3，可以检测出 2 位错或用来检测并纠正 1 位错。在设计系统时应事先确定相关要求：要么只要求发现错误而不纠正，则可以检测出 1 位出错或 2 位出错；要么只要求发现并自动纠正 1 位错。

能否做到检测两位错并纠正 1 位错呢？为了提高校验码的查错和纠错能力，需要进一步扩大码距。一种方案是：增加一个第 4 组，所有位(P_iA_i)都参加这组校验，则该组的检错信息 G_4 将能判别是一位错($G_4=1$)，还是两位错($G_4=0$)。如果是 1 位错，则由指误字译码输出将其变反纠正；如果是 2 位错，即 $G_4=0$，而 $G_3G_2G_1\neq0$，则只给出校验错信息，并不予以纠正。

在($k=4$，$r=3$)海明码中，待编有效信息为 4 位，而校验位为 3 位，冗余度较大，将降低外存的有效利用率。若增大 k，则 r 增加不多，冗余度降低，但 k 增大将导致硬件开销增加，一般要求 $k=8$，即以字节为单位进行海明编码。

海明校验至今仍是一种基本的校验方法，用于要求快速自动纠错的场合。

3)循环冗余校验

二进制信息位流沿一条线逐位在部件之间或计算机互换之间传送称为串行传送。循环冗余校验(cyclic redundancy check，CRC)能发现并纠正信息存储或传送过程中连续出现的多位错误，因此在磁表面存储器和多机通信中得到了广泛应用。CRC 所约定的校验规则是：如果校验码能被某一约定代码除尽，表明代码正确；如果除不尽，余数将指明出错位所在位置。

任意一串数码很可能除不尽，这时将产生一个余数。如果让被除数减去余数，势必能被约定除数除尽。但减法操作可能需要借位运算，难于用简单的拼装方法实现编码。因此采用一种模 2 运算，即通过模 2 减实现模 2 除，以模 2 加将所得余数拼接在被除数后面，形成一个能除尽的校验码。当然，在采用模 2 除时，对除数的选择是有条件的。

这里所讲的模 2 运算是一种以按位加减为基础的四则运算，不考虑进位和借位。注意，它和以 2 为模的定点运算是两个不同的概念。因此，模 2 加减即按位加减，也就是异或，可以用异或门实现。

待编码的信息是一串代码，可能是表示数值大小的数字，也可能是字符编码或其他性质的代码。在模 2 除中，暂将它视为数字，可用多项式来描述。定义待编信息（被除数）为 $M(x)$，约定的除数为 $G(x)$，因为它是用来产生余数的，所以 $G(x)$ 又称为生成多项式；所产生的余数为 $R(x)$，它相当于所配的冗余校验值。

(1)编码方法。

①将待编码的 k 位有效信息 $M(x)$ 左移 r 位，得 $M(x)\cdot x^r$。这样做的目的是空出 r 位，以便拼装将来求得的 r 位余数。

②选取一个 $r+1$ 位的生成多项式 $G(x)$，对 $M(x)\cdot x^r$ 作模 2 除运算。

$$\frac{M(x)\cdot x^r}{G(x)}=Q(x)+\frac{R(x)}{G(x)} \qquad (\text{模 2 除})$$

因为要产生 r 位余数，所以除数应为 $r+1$ 位。

③将左移 r 位的待编有效信息与余数 $R(x)$ 作模 2 加（减），即拼接为循环校验码。

$$M(x)\cdot x^r+R(x)=Q(x)\cdot G(x) \qquad (\text{模 2 加})$$

在按位运算中，模 2 加与模 2 减的结果是一致的，所以 $M(x)\cdot x^r-R(x)=M(x)\cdot x^r+R(x)$。$M(x)\cdot x^r$ 的末尾 r 位是 0，所以与余数 $R(x)$ 的加减实际上就是将 $M(x)$ 与 $R(x)$ 相拼接，拼接成的校验码必定能被约定的 $G(x)$ 除尽。本节将 $M(x)\cdot x^r+R(x)$ 称为循环校验码，但在许多磁表面存储器的记录格式中，常只将 $R(x)$ 称为校验码，简记为 CRC。

【例 2-22】 将 4 位有效信息 1100 编成循环校验码。

$M(x)=x^3+x^2$，　1100　　($k=4$)

$M(x)\cdot x^r=x^6+x^5$，　1100000　　($r=3$)

$G(x)=x^3+x+1$，　1011　　($r+1=4$)

$$\frac{M(x)\cdot x^3}{G(x)}=\frac{1100000}{1011}=1110+\frac{010}{1011}$$

```
          1110 ——— Q
1011 ) 1100000
       1011
       ----
        1110
        1011
        ----
        1010
        1011
        ----
        0010
        0000
        ----
         010 ——— R
```

(模2加)

$$M(x)\cdot x^3+R(x)=1100000+010=1100010 \qquad (\text{模 2 加})$$

将编好的循环校验码称为(7,4)码，即 $n=7$，$k=4$。

(2)译码与纠错。将收到的循环校验码用约定的生成多项式 $G(x)$ 去除。若编码无误，则余数为 0；若某一位出错，则余数不为 0。不同位出错，则余数不同，余数代码与出错位序号之间有唯一的对应关系。

通过【例 2-22】可求出其出错模式，即余数与出错位序号之间的对应模式，如表 2-4 所示。更换不同待测编码可以证明，出错模式只与码制和生成多项式有关，与编码代码无关。对于(7,4)循环码，表 2-4 具有通用性，可作为其判别依据。当然，对于其他码制或选用其他生成多项式的情况，出错模式可能不同。

表 2-4　(7,4)循环码的出错模式($G(x)=1011$)

A_1	A_2	A_3	A_4	A_5	A_6	A_7	余　数	出 错 位
1	1	0	0	0	1	0	0 0 0	无
1	1	0	0	0	1	1	0 0 1	7
1	1	0	0	0	0	0	0 1 0	6
1	1	0	0	1	1	0	1 0 0	5
1	1	0	1	0	1	0	0 1 1	4
1	1	1	0	0	1	0	1 1 0	3
1	0	0	0	0	1	0	1 1 1	2
0	1	0	0	0	1	0	1 0 1	1

表 2-4 中列举了 8 种情况：一种是正确编码，余数为 0；其余 7 种依次有 1 位出错，余数不为 0，与出错位序号有唯一的对应模式。深入研究发现一个有实用价值的规律，即如果有 1 位出错，那么用 $G(x)$除后得到一个不为 0 的余数，如果对该余数补 0 并继续除，那么各次余数将按表 2-4 中的顺序循环。例如，第 7 位 A_7 出错，余数为 001，补 0 后继续除，得余数 010，以后将依次得到余数 100，011，110，111 以及 101，然后又是 001，呈循环状态。这就是循环码名称的由来。

这一规律给我们的启示是：可以采取一种节省硬件的纠错办法，即只设置余数 101 的译码输出，对应于第 1 个出错位置；如果校验后发现余数不为 0，那么可以一边对余数补 0 继续进行模 2 除，同时将被检测编码循环左移一位；当出现余数 101 时，出错位也移至第 1 个位置，可用异或门将之变反纠正，或在由第 1 位移至第 7 位的途中予以纠正；继续移满一个循环，即共移 7 次，将得到一个纠正后的正确码字。

这样，就不必像海明校验那样为每一位提供纠正条件。当位数增多时，采用循环码能有效地降低硬件成本。循环码校验可用硬件逻辑实现，如在磁表面存储器中等；也可用软件实现，如在某些通信网中等。

(3)生成多项式的选取。并不是任何一个多项式都可以作为生成多项式的，从检错与纠错的要求出发，生成多项式应能满足一定要求，即任何一位发生错误都应使余数不为 0；不同位发生错误应当使余数不同；应满足余数循环规律。

对于使用者来说，可从有关资料上查到对应于不同码制的可选生成多项式。

通信系统中广泛使用的是下述两种标准：

CCITT(国际电报电话咨询委员会)推荐：$G(x)=x^{16}+x^{15}+x^2+1$。

IEEE(美国电气和电子工程师学会)推荐：$G(x)=x^{16}+x^{12}+x^5+1$。

2.2 定点数的表示

在计算机中，小数点不用专门的器件表示，而是按约定的方式标出。有两种方法约定小数点的存在：一种方法约定所有数据的小数点的位置固定不变，即定点表示；另一种方法约定小数点的位置是可以浮动的，即浮点表示。它们表示的数分别称为定点数和浮点数。

所有参与运算的数有两大类，即无符号数和有符号数。

2.2.1 无符号数的表示

所谓无符号数就是没有符号的数，其每一位均为有效值；而有符号数通常将最高位留作符号位。无符号数也称为非负数码(non-negative notation)，表示 0 或一个正数，相当于小数点位置隐含固定在最低位之后，如图 2-3 所示。n 位非负数码的数值范围为 $0\sim2^n-1$。

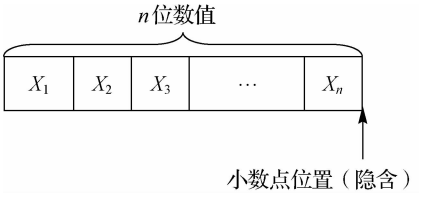


图 2-3　无符号数格式

2.2.2 有符号数的表示

对于有符号数，通常规定将符号位放在最高位，用“0”表示“正”，用“1”表示“负”，相当于比无符号数多一位。在有符号定点表示法中，有定点小数表示和定点整数表示两种。

1)有符号定点小数表示法

有符号定点小数小数点的位置固定在最高有效数位之前，符号位之后，记作 $X_0.X_1X_2\cdots X_n$，这个数是一个纯小数，如图 2-4 所示。定点小数的小数点的位置是隐含约定的，并不需要真正占据一个二进制位。

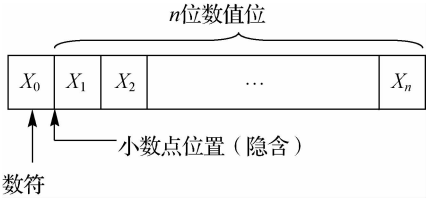
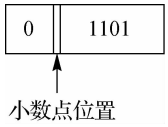
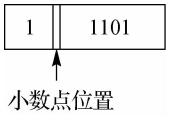


图 2-4　定点小数格式

例如，有符号小数＋0.1101 在机器中表示为：



有符号小数－0.1101 在机器中表示为：



在图 2-4 中，当 $X_0=0$，$X_1 \sim X_n$ 均为 1 时，X 为最大正数，即最大正数为$(1-2^{-n})$。

当 $X_0=1$ 时，表示 X 为负数，此时情况要稍微复杂一些，这是因为在计算机中带符号数既可用补码表示，也可用原码表示。如前所述，原码与补码所能表示的绝对值最大的负数是有区别的，所以原码和补码的表示范围有一些差别。设机器字长为 $n+1$ 位，则有如下结论：

原码定点小数表示范围为$-(1-2^{-n}) \sim (1-2^{-n})$。

补码定点小数表示范围为$-1 \sim (1-2^{-n})$。

分辨率为 2^{-n}。

2)有符号定点整数表示法

有符号定点整数小数点的位置隐含固定在最低有效数位之后，记作 $X_nX_{n-1}X_{n-2}\cdots X_0$，这个数是一个纯整数，如图 2-5 所示。

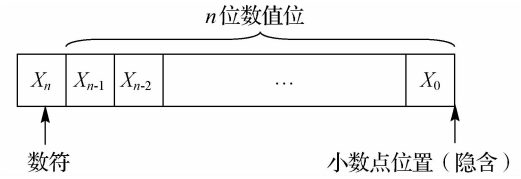
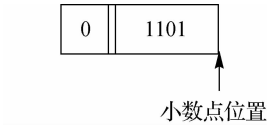
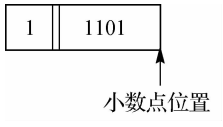


图 2-5　定点整数格式

例如，有符号整数＋1101 在机器中表示为



例如，有符号整数－1101 在机器中可表示为



原码定点整数的表示范围为$-(2^n-1) \sim (2^n-1)$。

补码定点整数的表示范围为$-2^n \sim (2^n-1)$。

分辨率为 1。

目前计算机常采用定点数来表示整数，即小数点右边没有数字，在本章后面所提到的定点数都是指定点整数。

在定点表示法中，参加运算的数以及运算的结果都必须在该定点数所能表示的数值范

围之内。若遇到绝对值小于分辨率的数，则被当做机器 0 处理；而大于最大正数和小于最小负数的数，统称为溢出，这时计算机将暂时中止运算操作，转而去进行溢出处理。

只有定点数表示的计算机称为定点计算机，在这种计算机中机器指令调用的所有操作数都是定点数。然而，实际需要计算机处理的数往往既有整数部分又有小数部分，对于定点计算机来说，这些数必须变为约定的定点数形式才能被处理，所以在编程时需要设定一个比例因子，把原始的数缩小成定点小数或扩大成定点整数后再进行处理，所得到的运算结果还需要根据比例因子还原成实际的数值。这显然很不方便，为此提出了浮点数的概念。

2.3 浮点数的表示

浮点数的主要特点是其小数点的位置根据需要而浮动，既能表示整数又能表示小数，并且小数部分的位数可以变化。本节首先讨论浮点数的格式和对某些特殊值的处理；接着研究浮点数的性质，如数据精度和数值范围等。

2.3.1 浮点格式

计算机中浮点数据表示的基本原理来源于十进制数中的科学计数法（scientific notation）。例如，在科学计数法中，电子的质量为 9×10^{-28} g，太阳的质量为 2×10^{33} g。可见，一个数在科学计数法中包含符号、小数、有效值（通常也称为尾数）和指数（通常也称为阶码），可写成如下形式：

$$N=M\times R^E$$

其中，R 为浮点数阶码的底，与尾数的基数相同，在实际的计算机中多取 2、4、8 和 16 等，以 $R=16$ 最为普遍。但在各种教科书中常约定尾数为二进制，相应地，取 R 为 2。E 和 M 都是带符号的定点数，E 叫做阶码（exponent），M 叫做尾数（mantissa）。在大多数计算机中，尾数为纯小数，常用原码或补码表示；阶码为定点整数，常用移码或补码表示。

浮点数的一般格式如图 2-6 所示，浮点数的底是隐含的，在整个机器数中不出现。阶码的符号位为 E_f，阶码的大小反映了在数 N 中小数点的实际位置；尾数的符号位为 M_f，它是整个浮点数的符号位，表示该浮点数的正负。浮点数的表示范围主要由阶码决定，有效数字的精度主要由尾数决定。

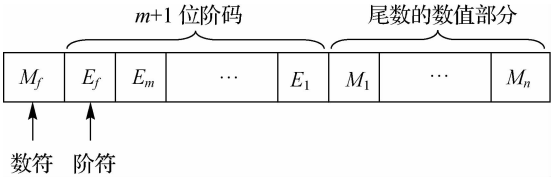


图 2-6 浮点数的一般格式

例如，某计算机的浮点格式为数符 1 位、阶码 8 位(补码)、尾数 23 位(补码)，则浮点数 17.5，即 0.10001×2^5 可以表示为

0	00000101	10001000000000000000000

2.3.2　规格化浮点数

浮点数的表示形式并不是唯一的，举例如下：

二进制数 $0.0001011=0.001011\times2^{-1}$

$=0.01011\times2^{-2}$

$=0.1011\times2^{-3}$

$=\cdots$

为了提高运算精度，充分利用尾数的有效数位，同时也使计算机实现浮点运算时有一个统一的标准形式，通常采取浮点数规格化形式，即规定尾数的最高数位必须是一个有效值。任何一个浮点数的规格化形式都是唯一的。

对于阶码的底为 R 的规格化浮点数，定义其尾数 M 的绝对值应在以下范围内：

$$1/R\leqslant M<1$$

若 $R=2$，则 $1/2\leqslant M<1$。当尾数用原码表示时，规格化尾数的最高数位总等于 1，为了扩大尾数的表示范围，有些机器在存储时把尾数的最高位隐含起来，这样就增加了一位尾数，从而提高了精度。当尾数用补码表示时，判断规格化尾数的方法是尾数的最高数位与符号位相反，即 $m_f\oplus m_1=1$。当 $1/2\leqslant M<1$ 时，应为 $0.1\times\cdots\times$ 形式，当 $-1\leqslant M<-1/2$ 时，应为 $1.0\times\cdots\times$ 形式。需要注意的是，此判断方法与定义有区别，即去掉了定义中的 $M=-1/2$ $(1.00\cdots0)$，加上了 $M=-1(1.00\cdots0)$。若 $R=4$，则 $1/4\leqslant M<1$，即尾数的最高两位不全为零的数为规格化数。规格化时，尾数左移两位，阶码减 1；尾数右移两位，阶码加 1。若 $R=8$，则尾数的最高 3 位不全为零的数为规格化数。规格化时，尾数左移 3 位，阶码减 1；尾数右移 3 位，阶码加 1。以此类推，可以得出 $R=16$ 或 2^n 时的规格化过程。

除了 0 以外，其他每个可能的数据均可以采用规格化表示，但由于 0 的所有有效值位均为 0，显然不能被规格化，因此要用一个特殊值表示 0，把 0 作为一种特殊情况处理。类似地，正无穷大和负无穷大也需要用一个特殊值来表示，并且也需要进行特殊处理。

另外，对一些非法操作，如$\frac{\infty}{\infty}$或对负数开平方等，称其结果为不存在的数，记为 NaN。NaN 也需要用一个特殊值来表示。编译器可以将值 NaN 设置为一个未初始化的变量。与 0 和无穷大类似，在浮点数算法中，NaN 也要求进行特殊处理。

注意，阶码包含符号位，可以用补码表示，但更有优势的实用方法是采用移码(biasing)表示，它能使浮点数的加减法运算更加方便。

2.3.3　移码(增码)表示法

众所周知，小数在进行加减运算时，应先将小数点对齐，反映在浮点数加减运算中，就是将两数的阶码调整为相同，称为对阶。为了更直观地比较正负阶码的大小，浮点数可用移码

表示阶码，定义如下：

设定点整数移码序列为 $X_0X_1X_2\cdots X_m$，则

$$[X]_{移}=2^m+X \qquad -2^m\leqslant X<2^m \tag{2-10}$$

其中，X 是阶码的真值，2^m 是符号位的位权，这相当于将真值 X 沿数轴正向平移量 2^m，所以称其为移码，如图 2-7 所示。或者说将 X 增加一个量 2^m，因此又称其为增码。

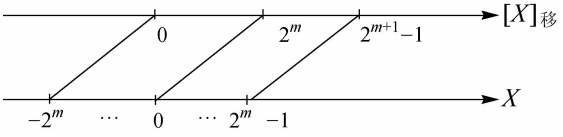


图 2-7　移码在数轴上的表示

例如，某浮点数阶码共 8 位，其表示范围为 $-128<X<127$，现用移码可以表示为

$$[X]_{移}=2^7+X$$

真值、移码、补码之间的对应关系如表 2-5 所示。移码符号位为 0 时是负值，为 1 时是正值，这与其他码制正好相反。除符号位外，其余各位与补码相同。这是由于移码平移了 2^7，而补码平移了 2^8。

通过分析表 2-5，可以得出移码如下一些性质。

(1)表中$[X]_{移}$相当于把真值映射到 0～256 的正数域。若将移码看做无符号数，则移码的大小反映了真值的大小，这样便于对两个浮点数的阶码大小进行比较。

(2)最高位为符号位，表示形式与原码和补码相反，1 表示正，0 表示负。

(3)移码与补码的关系：$[X]_{补}=2^{n+1}+X=2^n+2^n+X=2^n+[X]_{移}$；从形式上看，$[X]_{移}$与$[X]_{补}$除符号位相反外，其余各位均相同。

(4)在移码表示中，0 有唯一的编码，即$[+0]_{移}=[-0]_{移}=100\cdots0$。

(5)$[X]_{移}$为全 0 时，表明阶码最小，即绝对值最大的负数，为-2^n。

表 2-5　真值、移码、补码对照表

真值 X （十进制）	真值 X （二进制）	$[X]_{移}$	$[X]_{补}$
−128	−10000000	00000000	10000000
−127	−01111111	00000001	10000001
−126	−01111110	00000010	10000010
—	—	—	—
—	—	—	—
—	—	—	—
−2	−00000010	01111110	11111110
−1	−00000001	01111111	11111111

续表

真值 X （十进制）	真值 X （二进制）	$[X]_{移}$	$[X]_{补}$
0	00000000	10000000	00000000
+1	00000001	10000001	00000001
+2	00000010	10000010	00000010
—	—	—	—
—	—	—	—
—	—	—	—
+126	11111110	11111110	01111110
+127	11111111	11111111	01111111

2.3.4　浮点数据特性

浮点数有几种固有的特性，包括精度、间距和范围。

1）精度

顾名思义，精度（precision）表示浮点数的精确性，它被定义为尾数的位数。若一个计算机规定浮点数的尾数为 8 位，则它的精度为 8 位。尾数的位数越多，CPU 的精度就越高，它表示的数据就越精确。许多计算机中有两种浮点数格式，即单精度浮点数和双精度浮点数。双精度浮点数的位数是单精度浮点数位数的 2 倍。

2）间距

间距（gap）为两个相邻值的差的绝对值，它的大小由阶值的大小决定。例如，对于一个有 8 位尾数的浮点数 0.10111010×2^3，它的 2 个相邻值分别为 0.10111011×2^3 和 0.10111001×2^3，间距为 $0.00000001\times2^3=2^{-5}$。实际上浮点数 X 的间距可以表示为 $2^{(X-\text{precision})}$。

3）范围

范围（range）由浮点数格式所能表示的最大值和最小值决定，它与阶码的底 R 有关，也与阶码和尾数的位数以及采用的机器数表示形式有关。例如，采用图 2-6 所示的浮点数格式，即阶码为 $m+1$ 位（含 1 位阶符），移码表示，以 2 为底；尾符为 1 位，尾数为 n 位，规格化且补码表示，则表 2-6 列出了此浮点数的几种典型值。其中，为了便于阅读，将浮点代码的数符与尾数写在了一起（机器中的数符是在最高位的），还在阶码与尾数之间加注了分号，并在数符后标注了小数点（在机器中分号与小数点并不存在）。整个数据范围在数轴上的表示如图 2-8 所示。

表 2-6　浮点数的典型值

数　　据	浮点数代码		真　　值
	阶码	尾数	
最大正数 A	111…1	0.11…1	$(1-2^{-n})2^{2^m-1}$
非 0 最小正数 a	000…0	0.10…0	$\frac{1}{2}\cdot 2^{-2m}$
绝对值最大负数 B	111…1	1.00…0	$(-1)\cdot 2^{2m-1}$
绝对值最小负数 b	000…0	1.01…1	$\left(-\frac{1}{2}-\frac{1}{2^n}\right)\cdot 2^{2m-1}$

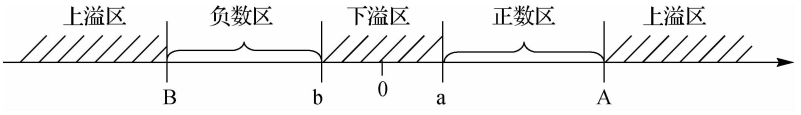


图 2-8　规格化浮点数在数轴上的表示范围

为了说明这些性质，现考虑一种浮点数格式，即有 1 位符号位，8 位阶码且偏移量为 128，23 位尾数。这种格式的精度为 23 位，范围近似为 $-1.7\times10^{38}\sim+1.7\times10^{38}$。它的间距由实际的浮点数决定，例如，对于 0.1×2^{127}，间距为 $2^{(127-23)}=2^{104}$；对于 0.1×2^{-128}，间距为 $2^{(-128-23)}=2^{-151}$。

当浮点数操作所产生的结果不能存于计算机的浮点寄存器时，就发生了溢出，浮点数的溢出分为上溢和下溢。在有 8 位尾数和 4 位阶码（偏移量为 8）的浮点数例子中，浮点数 0.1×2^6 和 0.1×2^5 相乘将得积 $0.01\times2^{11}=0.1\times2^{10}$，积的阶值大于该格式所允许的最大阶值 7，因此产生了上溢。上溢可以为正或为负，这由上溢值的符号决定。上溢值可以处理为 $\pm\infty$ 或 NaN，或像定点数算法的溢出一样置上溢标志位为 1。

当结果在 0 到最小正值或最小负值之间时，将产生下溢。例如，同样在有 8 位尾数和 4 位阶码（偏移量为 8）的浮点数例子中执行乘法 $(0.1\times2^{-6})\times(0.1\times2^{-5})$，所得结果为 0.1×2^{-12}，它在 $0\sim0.1\times2^{-8}$ 之间，因此产生下溢。下溢也可以为正或为负，下溢值可以处理为 0 或者设置下溢标志位。

4）舍入方法

不管尾数的位数有多大，它都不可能精确地表示每一个可能的数，即某些操作产生的结果，其尾数的位数可能太多而不能放入 CPU 的寄存器中。例如，两个 8 位尾数的浮点数相乘，得到积的尾数为 16 位。因此必须将该值进行转换，使之能放入 8 位的尾数寄存器中，该过程即为舍入处理。

舍入处理的目标是使舍入后的结果尽可能接近实际值。例如，将值 0.10110101 舍入为小数点后 4 位，得到值 0.1100，这种舍入方法称为就近舍入法，也称无偏舍入法，它的最大误差为 ±0.5LSB（LSB 为舍入后结果的最低尾数值）。

就近舍入法是最常使用的舍入方法。除此之外还有以下几种常用的舍入方法。

（1）截断法。通过截断实际值的多余位来实现。这种方法的硬件实现非常简单，但它的最大误差将增大到 ±LSB。

(2)向+∞舍入法。结果向正无穷大方向舍入。所有值都被舍入为下一个可能值,负数值的结果是截去了数值的多余位,正数值舍入后的结果为下一个较大的尾数,当丢弃的位均为 0 时该值保持不变。例如,-0.10111111、+0.10110000 和+0.10110001 的舍入结果分别为-0.1011、+0.1011 和+0.1100。

(3)向-∞舍入法。结果向负无穷大方向舍入。负数值舍入后的结果为下一个较小的尾数,正数值舍入只要截去多余位即可。例如,-0.10111111、+0.10110000 和+0.10110001 舍入后的结果分别为-0.1100、+0.1011 和+0.1011。

表 2-7 给出了 8 位数值在不同的舍入方法下舍入为 4 位数值的舍入结果。

表 2-7　不同舍入方法的比较

数　值	就近舍入法	截 断 法	向+∞舍入法	向-∞舍入法
.01101001	.0111	.0110	.0111	.0110
-.01101001	-.0111	-.0110	-.0110	-.0111
.10000111	.1000	.1000	.1001	.1000
-.10000111	-.1000	-.1000	-.1000	-.1001
.10000000	.1000	.1000	.1000	.1000

除了截断法外,其他舍入法都要求在最终的表示之外增加额外的几位,通常只要 1、2 位就足够了。这些位的最高位称为舍入位,次高位称为保护位。在后面的浮点数运算的算法中,将看到这些位是对结果的补充。第三位称为粘滞位,通常用于浮点数的乘法运算中,初始化为 0,在 1 被右移出保护位时置 1,不同于舍入位和保护位,粘滞位置 1 后,值就不再改变。在某些算法中粘滞位可能取代保护位。

2.3.5　实例:IEEE 754 浮点标准

20 世纪 70 年代末,计算机制造商在许多计算机上实现了浮点操作,但由于采用了不同的方法,导致同一个程序在不同的计算机上运行可能产生不同的结果,这显然是不能接受的。为此,IEEE 提出了 IEEE 754 浮点标准。目前该标准应用于所有具有浮点功能的 CPU 中。本小节将简单介绍该标准的格式,同时介绍非规范数,这是 IEEE 754 标准采用的一个新概念。

1)格式

IEEE 754 标准包括两种精度的浮点数格式。一种是单精度浮点数,共有 32 位,即 1 位符号、8 位阶码和 23 位尾数。该尾数在小数点的左边包含一个暗藏的 1(特殊值和下一小节将要介绍的非规范数除外)。不同于前面所讨论的浮点数格式,该格式的规格化尾数的绝对值在 1～2 之间。其阶码的偏移量为 127,阶值在 -126～+127 之间。阶码 00000000(-127)和 11111111(+128)用来表示特殊值。

表 2-8 给出了+19.5(二进制表示为 10011.1 或 1.00111×2^4)的单精度浮点数格式。注意,先导 1 并没有在有效值中给出,它是暗藏在该标准中的。另外,由于要加上偏移量 127,因此阶值 4 被表示为 10000011,即 131。

另一种是双精度浮点数,共有 64 位,即 1 位符号、11 位阶码和 52 位尾数。与单精度浮

点数相同，对于大多数数值，其尾数有一个暗藏的先导 1。阶码的偏移量为 1 023，阶值范围为－1 022 ～＋1 023，阶码－1 023 和＋1 024 同样用来表示特殊值。表 2-9 给出了＋19.5的双精度浮点数格式。同样地，阶码等于阶值加上偏移量，即 4＋1 023＝1 027。

单精度浮点数和双精度浮点数中都规定了 0、±∞ 和 NaN 的表示形式。0 的尾数和阶码均为 0。无穷大的尾数为 0，阶码的所有位均为 1，而符号位表明是＋∞ 还是－∞。NaN 的有效值为一个非 0 的任意值，阶码的所有位均为 1。

表 2-8　IEEE 754 标准单精度浮点数格式

数　值	符　号	尾　数	阶
＋19.5	0	00111000000000000000000	10000011
0	0	00000000000000000000000	00000000
±∞	0 或 1	00000000000000000000000	11111111
NaN	0 或 1	非 0 的任意值	11111111
非规范数	0 或 1	非 0 的任意值	00000000

表 2-9　IEEE 754 标准双精度浮点数格式

数值	符号	尾数	阶
＋19.5	0	00111000000000000000000000000000000000000000000000000000	10000000011
0	0	00000000000000000000000000000000000000000000000000000000	00000000000
±∞	0 或 1	00000000000000000000000000000000000000000000000000000000	11111111111
NaN	0 或 1	非 0 的任意值	11111111111
非规范数	0 或 1	非 0 的任意值	00000000000

前面介绍了 4 种舍入方法，即就近舍入法、截断法、向＋∞舍入法和向－∞舍入法。IEEE 754 标准中默认的舍入方式为就近舍入法，但用户也可以选择其他舍入方式。因此，对于遵循 IEEE 754 标准的两个处理器，如果采用不同的舍入方式，那么对同一个计算产生的结果可能稍有不同。

2)非规范数

在 IEEE 754 标准中，大部分浮点数都是规格化浮点数，即它们的尾数都有一个暗藏的先导 1。不过除了前面所讨论的几种特殊值外，还有一种浮点数的尾数不用暗藏的 1，称其为非规范数。非规范数的阶码为全 0，尾数为一个非 0 值(见表 2-8 或表 2-9)。

规格化浮点数最小正值的阶码为 00000001(－126)，尾数为 00000000000000000000000(1.0，存在暗藏的 1)，因此该值为 1.0×2^{-126}。使用非规范数，计算机可以表示更小的非 0 值。非规范数的阶码为 00000000，此时表示阶码－126。对于非规范数，尾数不用暗藏的 1，因此 23 位的尾数表示一个完整的尾数。例如，有效值为 10000000000000000000000 的非规范数表示 $0.1\times2^{-126}=2^{-127}$。使用非规范数单精度浮点数所能表示的最小正值为 $00000000000000000000001\times2^{-126}=2^{-149}$；而双精度浮点数所能表示的最小正值为 $2^{-1\,074}$。

习　题　2

(1)将下列各十进制数分别转换为二进制、八进制、十六进制和 BCD 码形式。

①2 139。　　②84.675。

(2)将下列各二进制数分别转换为十进制、八进制和十六进制形式。

①10110.1011。　　②1101110.01101。

(3)给定下列十六进制数,若将此数分别视为无符号数、原码、补码、反码和移码表示,写出其对应的十进制整数(符号占一位)。

00H,50H,7FH,80H,85H,FEH,FFH。

(4)某定点小数字长为 16 位,含 1 位符号,原码表示,分别写出下列各小题的二进制代码和十进制真值。

①非零最小正数。　　②最大正数。

③绝对值最小非零负数。　　④绝对值最大负数。

(5)某定点小数字长为 16 位,含 1 位符号,补码表示,分别写出下列各小题的二进制代码和十进制真值。

①非零最小正数。　　②最大正数。

③绝对值最小非零负数。　　④绝对值最大负数。

(6)试比较定点带符号数在计算机中的 4 种表示方法。

(7)写出下列字符串的 ASCII 码。

①“You Are Students”。　　②“34.78”。

(8)设计一个可对 4 位二进制码进行奇校验的校验电路。

(9)为什么在海明码纠错过程中,指误字 $G_3G_2G_1$ 的状态可以指出编码中错误的信息位?

(10)有一个(7,4)码,其生成多项式为 $G(x)=x^3+x+1$,试求出代码 1001 的循环冗余校验码。

(11)若浮点数格式为尾数部分取 10 位原码表示,阶码部分取 4 位移码表示,阶符和数符各取 1 位,试将下列十进制真值变换为要求的浮点数格式。

①31.125。　　②−123.5。

③13/128。　　④−13/102。

(12)某浮点数字长为 32 位,其中阶码 8 位(含 1 位阶符),以 2 为底,补码表示;尾数为 24 位(含 1 位数符),补码表示。现有一浮点数为 AC5A3E00H,则它所表示的二进制真值是多少?

(13)某浮点数字长为 16 位,其中阶码 6 位(含 1 位阶符),补码表示,以 2 为底;尾数 10 位(含 1 位数符),补码表示。分别写出下列各小题的二进制代码和十进制真值。

①最大正数。　　②非零最小正数。

③最大规格化正数。　　④最小规格化正数。

⑤绝对值最小的非零负数。　　⑥绝对值最大的负数。

⑦绝对值最小的规格化负数。　　⑧绝对值最大的规格化负数。

(14)某浮点数字长为32位,其中阶码8位(含1位阶符),尾数24位(含1位数符),当阶码的底分别为2和16时,回答下述问题。

①2和16在浮点数中如何表示?

②当阶码和尾数均为补码表示,且尾数采用规格化形式时,两种情况下所能表示的最大正数真值和非零最小正数真值分别为多少?

③数的表示范围有何不同?

(15)某浮点数字长为16位,其中阶码6位(含1位阶符),移码表示,以2为底;尾数10位(含1位数符),补码表示,计算

①浮点数为83BEH的十进制数值。

②此浮点格式的规格化表示范围。

(16)设浮点数字长为32位,欲表示十进制数±60 000,在保证最大精度要求条件下,除阶符、数符各取1位外,阶码和尾数各取几位?按这样的分配,该浮点数溢出的条件是什么?

(17)写出下列十进制数的IEEE 754单精度格式。

①9。　　②−5/32。

(18)写出下列十进制数的IEEE 754双精度格式。

①5/32。　　②−6.125。

第 3 章 指令集结构

不同系列的计算机具有不同的指令集结构(instruction set architecture,ISA),指令集结构是依据计算机中 CPU 的硬件特点而设计出来的,与 CPU 的性能有着密不可分的关系。为了实现高性能,人们采用各种先进技术来设计 CPU,但最终都是通过设计一套优质的指令集来展现计算机的强大功能。

指令集结构的功能与该计算机硬件结构的复杂性往往成正比。计算机的指令越丰富,功能就越强,软件运行的效率就会越高,但该机的硬件结构也就越复杂,设计起来也就越困难,代价也越高;反之,若该机的指令集结构提供的功能简单,则硬件结构和设计就简单,代价也低,但同时软件运行的效率也降低了。因此,设计一台计算机的指令集结构时,必须考虑多种因素,合理地在机器的软件和硬件中进行功能分配。

本章将从硬件设计的需要出发,对指令集结构的一些基本概念,如指令格式、指令种类、操作码的编码、寻址方式等进行讨论。

3.1 指令集结构概述

指令集结构是计算机的硬件和软件的主要分界面,从机器语言程序员(或汇编语言程序员)角度来看,计算机的指令集结构是该机具有的基本属性和所能提供的基本功能;从硬件设计者角度来看,计算机的指令集结构是硬件设计的依据和目的。

3.1.1 指令集结构的定义

指令集结构包含了一台计算机所能执行的全部指令集,即 CPU 所能执行的所有机器语言(或汇编语言)的指令集合。但指令集结构不仅包含指令集,还要考虑寄存器的设置、主存储器的结构、机器直接支持的数据类型等。ISA 的主要内容包括如下几个方面:

(1)操作指令表(operation repertoire)。操作指令表指明应提供多少操作和什么样的操作,操作将是何等复杂。

(2)数据类型(data types)。数据类型指对几种数据类型完成操作。

(3)指令格式(instruction format)。指令格式包括指令的(位)长度、地址数目、各个字段的大小等。

(4)寄存器(registers)。寄存器包括能被指令访问的 CPU 寄存器的数目以及它们的用途。

(5)寻址(addressing)。寻址用于指定操作数地址的产生方式。

上述因素是紧密相关的,设计指令集结构时必须予以综合考虑。

3.1.2 指令集结构设计原则

指令集的设计是计算机设计中最有趣、也最受关注的方面。指令集的设计是一件很复杂的事情,因为它影响计算机系统的诸多方面。指令集定义了 CPU 应完成的功能,因此它对 CPU 的实现有显著的影响。指令集亦是程序员控制 CPU 的方式,所以设计指令集时必须考虑程序员的要求。

不同类型的计算机有各具特色的指令集,由于计算机的性能、机器结构和使用环境不同,指令集的差异也是很大的。因此,试图给指令集确定一个统一的衡量标准是很困难的,主要考虑的因素有 3 个,即速度、价格和灵活性。设计指令集时,在功能方面的最基本要求包括指令集的完整性、规整性、高效性和兼容性。

1)指令集的完整性

指令集的完整性是指作为通用计算机所应该具备完备和丰富的基本指令种类,本节将讨论一些基本的指令种类。

2)指令集的规整性

指令集的规整性设计是硬件设计(如 VLSI 技术等)和软件设计(如编译程序等)的需要。其含义是指尽可能减少、甚至避免出现任何例外情况和特殊用法,以使所有运算部件都能对称、均匀地在所有数据存储单元(主存储器、通用寄存器和堆栈)之间进行操作,对所有数据存储单元都能同等对待,无论是操作数还是运算结果都可以无任何约束地存放在任意数据存储单元中。

规整性主要包括对称性和均匀性。对称性是指各种与指令集有关的数据存储设备的使用、操作码的设置等都要对称。例如,所有通用寄存器都要同等对待,这在目前的许多计算机系统中都没有做到,往往隐含规定某一个或某几个通用寄存器有特殊的用途。在操作码的设置上,如果设置有 A—B 指令,那么也应该设置 B—A 指令。对于包含两个地址码的指令,由于运算结果要破坏掉其中一个源操作数,因此最好设置两种,如 A+B→A 和 A+B→B 等。均匀性是指对于各种不同的数据类型、字长、操作种类和数据存储设备(通用寄存器、主存储器、堆栈及输入/输出设备等),指令的设置要同等对待。例如,某机器有 5 种数据表示(定点数、逻辑数、浮点数、十进制数、字符串),4 种字长(单字长、双字长、半字长、字节),8 种数据存储设备的有效排列(通用寄存器、主存储器、堆栈及它们之间有用的排列),则在设计加法指令时,指令种类应该有 5×4×8=160 种两地址加法指令。如果再考虑对称性的要求和不同寻址方式的要求等,那么指令种类还要增加很多,这实际上很难做到,也是不现实的。

因此，在设计指令集时对规整性的要求必须有所选择。在 RISC（精简指令集计算机）体系结构中，规定运算型指令都在通用寄存器内进行操作，即使在 CISC（复杂指令集计算机）体系结构中，如果采用二地址通用寄存器结构，也通常规定两个地址中必须有一个是通用寄存器。另外，对于逻辑数、十进制数、字符串等数据表示，双字长、半字长和字节等数据长度，可以适当减少指令种类。这样，可以把加法指令的种类压缩在 10 种之内。另外，如果采用自定义数据表示方式，那么每个数据只要带有几个标志位，规整性问题就不难解决了。

3）指令集的高效性

高效性是指指令的执行速度要快，使用频度要高。在 RISC 体系结构中，大多数指令都能在一个节拍内完成，而且只设置那些使用频度高的指令。而在 CISC 体系结构中，有些指令的执行速度非常慢，需要几十个节拍才能完成，甚至是不固定节拍的（由所使用的数据决定指令执行所需要的节拍数，如检索、匹配、数据块搬家等指令）。在设置这些低速度指令时要特别慎重，要考虑是用软件的一段子程序来实现，还是用硬件的一条指令来实现。对于那些使用频度比较低的指令，也要尽量少设置。在考虑指令的实现方法时，对于那些比较复杂但又必须设置的指令，可以采用微程序来实现，以减少硬件的复杂程度。总之，指令集的设计要进行大量的统计和模拟工作，要在反复比较的基础上才能最后确定下来。

4）指令集的兼容性

如果在一个机器上设计的程序能够在另外一台机器上执行而无需对程序进行任何修改，就称它们的指令集是兼容的。兼容性是计算机系统的生命力之所在。没有兼容性，大量的系统软件和各种应用软件就无法继承，计算机也就没有了市场。因此，在设计指令集时，兼容性是必须考虑的因素。

3.2　指 令 格 式

CPU 所执行的操作是由它运行的指令所决定的，在每一条指令中都必须包含 CPU 运行该指令所要求的全部信息。CPU 执行一条指令的操作过程如图 3-1 所示。

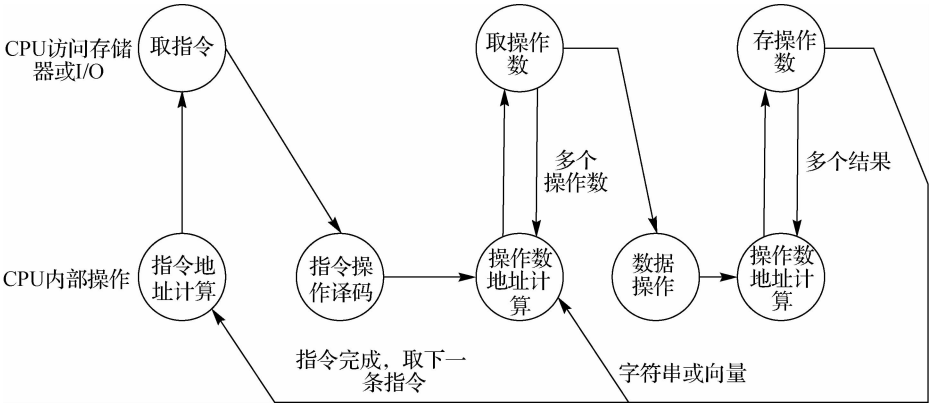


图 3-1　指令周期状态图

一条指令应包含以下信息。

(1)操作码。操作码用来指明 CPU 执行什么样的操作,即操作的类型,在机器内部采用二进制编码。

(2)源操作数地址。源操作数地址用来指明 CPU 运算过程中要用到的操作数或操作数的地址。

(3)目的操作数地址。目的操作数地址告诉 CPU 将运算结果存放到何处。

(4)下一条指令的地址。下一条指令的地址指当前指令运行结束后,CPU 紧接着执行的下一条指令在内存单元中的地址。

指令的基本格式可表示为:

操作码 OP	地址码 A

操作码的长度取决于共有多少类不同的操作,即一个机器的指令集中所包含的指令的数量。地址码的位数取决于地址的结构。另外,在指令中通常还应给出操作数的数据类型、寻址方式等附加信息。

下面先介绍操作码编码及优化的方法,然后再从地址结构的角度介绍几种指令格式。

3.2.1 操作码

指令集中的每一条指令都有一个唯一确定的编码(操作码),指令不同,其操作码也不相同。为了能表示整个指令集中的全部指令,指令的操作码字段应当具有足够的长度。例如,某一机器的指令集共含有 n 条指令,则指令中操作码字段的长度至少应为 $\log_2 n$ 位。

一般地,在确定操作码的长度时需要考虑两方面的问题:一方面希望用尽可能短的操作码字段来表达全部的指令;另一方面则希望硬件的实现尽可能简单。而要使这两方面都取最优是不可能的,侧重点不同,从而指令操作码的编码就不同。一般情况下可以分为 3 种,即固定长度编码、Huffman 编码和扩展编码。

1)固定长度编码

固定长度编码是一种最简单的编码方法,操作码字段的位数是固定的,从而使指令非常规整,硬件的译码也很简单。这种编码方法广泛应用于字长较长的大中型计算机、超级计算机以及 RISC 中。例如,IBM 370 机和 VAX-11 系列机的操作码都是 8 位固定长度。固定长度编码的主要缺点是信息的冗余极大,从而使程序的总长度增加。

2)Huffman 编码

Huffman 编码是 1952 年由 Huffman 首先提出的,其基本原理是指令的操作码部分采用变长编码,即对于使用频率很高的指令,采用较短的操作码位数进行编码;而对于使用频率很低的指令,则采用较长的操作码位数进行编码,从而缩短了程序的总长度。

采用 Huffman 编码时,必须先知道各种指令在程序中出现的概率,这通常可以通过对已有的一些典型程序进行统计而得到。采用 Huffman 编码的操作码的平均长度计算公式为:

$$H = \sum_{i=1}^{n} p_i \cdot l_i$$

其中，p_i 表示第 i 种操作码在程序中出现的概率，l_i 表示第 i 种操作码的 Huffman 编码的长度，操作码总共有 n 种。

下面通过一个例子来说明采用 Huffman 编码优化操作码的具体做法，并计算操作码的平均长度。

【例 3-1】　假设一台模型计算机共有 7 种不同的指令，各种指令在程序中出现的概率如表 3-1 所示，试给出各指令的操作码的 Huffman 编码，并计算操作码的平均长度。

表 3-1　模型机的各种指令的出现概率

指令序号	I_1	I_2	I_3	I_4	I_5	I_6	I_7
出现的概率	0.30	0.25	0.15	0.10	0.10	0.05	0.05

解：首先，把所有 7 条指令按照操作码在程序中出现概率值的大小自左向右排列好，每条指令作为一个结点；然后，选取两个概率值最小的结点合并成一个新结点，新结点概率值为两者之和，并把这个新结点插入其他还没有合并的结点中间得到一个新的结点集合；再在新的结点集合中选取两个概率最小的结点进行合并，如此继续进行下去，直至全部结点都合并完毕，最后得到只剩一个根结点的结点集合，根结点的概率值为 1。整个过程如图 3-2 所示。

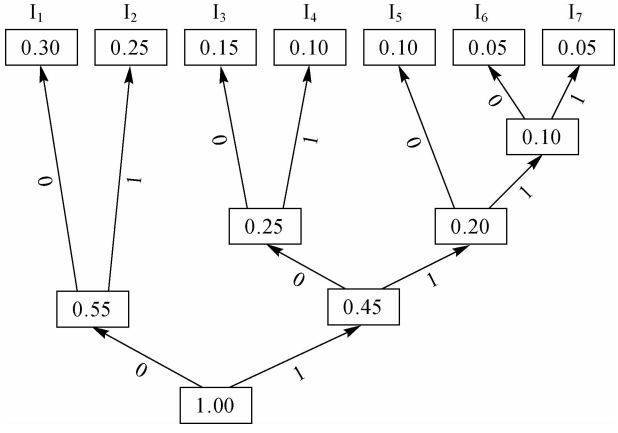


图 3-2　Huffman 代码树

从图 3-2 中可以看到每个结点都有两个分支，分别在每个结点的左分支上写代码“0”，在结点的右分支上写代码“1”。如果要得到某一条指令的 Huffman 编码，可以从根结点开始，沿箭头所指方向到达属于该指令的概率结点，将沿线所经过的代码组合起来即可。参照图 3-2 的 Huffman 代码树可以写出模型机各条指令的 Huffman 编码，如表 3-2 所示。

表 3-2　模型机各指令的 Huffman 编码

指令序号	I_1	I_2	I_3	I_4	I_5	I_6	I_7
出现的概率	0.30	0.25	0.15	0.10	0.10	0.05	0.05
Huffman 编码	00	01	100	101	110	1110	1111

应当指出，采用上述方法形成的编码并不是唯一的，只要将任意一个二叉结点上的“0”和“1”交换，就可以得到一组新的编码。然而，无论怎样交换，操作码的平均长度都是唯

一的。

采用 Huffman 编码所得到的操作码的平均长度为

$$H=\sum_{i=1}^{7}p_i\cdot l_i$$
$$=0.30\times2+0.25\times2+0.15\times3+0.10\times3+0.10\times3+0.05\times4+0.05\times4$$
$$=2.45$$

即平均码长为 2.45 位，比固定码长(3 位)减少了 18%，这个方案也满足译码唯一性。

3)扩展编码

由于操作码字段的位数不固定，增加了指令译码和分析的难度，使控制器的设计复杂化，因此，最常用的非规整型编码(非定长编码)方式是扩展操作码法。

扩展编码的基本思想为：在指令长度固定的情况下，使操作码的长度随地址数的减少而增加，不同地址数的指令可以具有不同长度的操作码。这样既能充分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，从而使它能表示更多的指令。

例如，设某模型机的指令长度为 16 位，操作码字段为 4 位，有 3 个 4 位的地址码字段，其格式如图 3-3 所示。

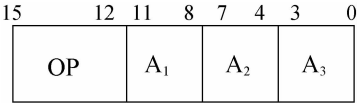


图 3-3　某模型机指令格式

按照固定长度编码的方法，4 位操作码字段只可能表示 16 条不同的三地址指令。现假设指令集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，利用扩展操作码可以在指令长度不变的情况下，使指令的总数远远大于 16 条。扩展的方法如下。

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

(2)8 位操作码字段的编码 11110000～11111110 可以定义 15 条二地址指令，留下 11111111 作为扩展窗口，与 A_2 组成一个 12 位的操作码字段。

(3)12 位操作码字段的编码 111111110000～111111111110 可以定义 15 条一地址指令，扩展窗口为 111111111111，与 A_3 组成一个 16 位的操作码字段。

(4)最后，16 位操作码字段 1111111111110000～1111111111111111 给出 16 条零地址指令。

这样，在指令长度保持不变的情况下，指令集中就包含了 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，总共 61 条指令。

按照指令集的要求，扩展操作码的组合方案可以有很多种，但有一点要注意：各条指令的操作码一定不能相同，即不同的指令有不同的操作码，而且各类指令的格式安排应统一规整。

3.2.2　地址码

地址码用来指出该指令的源操作数的地址、结果的地址和下一条指令的地址。这里的“地址”可以是主存的地址、寄存器的地址或 I/O 设备的地址。

下面以主存地址为例，分析指令的地址码字段。

1)四地址指令

四地址指令的格式为：

操作码 OP	第一操作数 A_1	第二操作数 A_2	结果 A_3	下一条指令地址 A_4

功能描述如下：

$A_3 \leftarrow (A_1)OP(A_2)$

$(A_4)=$ 下一条指令地址

其中，A_i 表示该地址对应的存储单元，(A_i) 表示地址为 A_i 的存储单元中的内容，下同。$(i=1,2,3,4)$。

这种格式的优点是直观，下一条指令的地址明显，不需要转移指令；缺点是指令太长，如果每个地址为 16 位，整个地址码字段就要长达 64 位；另外因为直接给定了后继指令地址，程序将不能根据操作结果灵活转移，所以这种格式实际上较少被应用。

2)三地址指令

因为程序中的大多数指令是按顺序从主存中逐条取出并执行的，而程序计数器(program counter，PC)既能存放当前欲执行指令的地址，又具有计数功能，因此它能自动加 1(设该指令只占一个主存单元)，以形成下一条指令的地址。这样，指令中就不必再显式给出下一条指令的地址了，三地址指令格式为：

操作码 OP	第一操作数 A_1	第二操作数 A_2	结果 A_3

功能描述如下：

$A_3 \leftarrow (A_1)OP(A_2)$

$PC \leftarrow (PC)+1$

这种格式虽然省去了一个地址，但指令仍比较长，所以只在字长较长的大中型机中使用。

3)二地址指令

对于三地址指令，在执行完毕后，主存中的两个操作数均不被破坏，可供再次使用。然而，通常并不需要完整地保留两个操作数。如果让第一操作数地址同时兼作存放结果的目的操作数的地址(目的地址)，即得二地址指令格式为：

操作码 OP	第一操作数 A_1	第二操作数 A_2

功能描述：

$A_1 \leftarrow (A_1)OP(A_2)$

$PC \leftarrow (PC)+1$

此时，A_1 既是源操作数的地址，又是目的操作数的地址；A_2 为另一个源操作数的地址。

二地址指令在计算机中得到了广泛应用，但是在使用时有一点必须注意，即指令执行后 A_1 中原来存储的内容已经被新的运算结果所替换。

4)一地址指令

一地址指令的格式为：

操作码 OP	第一操作数 A_1

一地址指令可能有如下两种情况。

(1)单操作数运算指令。如“加 1”、“减 1”、“求反”等操作,由于这类指令仅需要一个操作数,所以地址 A_1 既是源操作数地址,又是存放结果的地址。

功能描述如下:

$A_1 \leftarrow OP(A_1)$

$PC \leftarrow (PC)+1$

(2)隐含一个操作数。指令中只给出一个源操作数地址 A_1,而隐含了 CPU 内部的累加器 AC,AC 既提供另一个源操作数的地址,同时又作为运算结果的存放地址。

功能描述:

$AC \leftarrow (AC)OP(A_1)$

$PC \leftarrow (PC)+1$

5)零地址指令

零地址指令中只有操作码字段,没有地址码字段,其格式为:

操作码 OP

只有操作码字段的指令可能是不需要操作数的指令,如“停机”、“空操作”、“清除”等控制类指令;也可能是堆栈指令,在堆栈计算机中参加运算的操作数存放在堆栈中,运算结果也存放在堆栈中,因此,无需在指令中显式地指出操作数的地址。

指令中地址个数的选取要考虑许多因素。从缩短程序长度、方便用户使用、增加操作并行度等方面来看,选用三地址指令格式比较好;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,但指令最长,而分别用二地址指令、一地址指令和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。图 3-4 给出了用不同地址长度的指令来计算表达式 X=(A+B×C)÷(D−E)的程序。

指　令	功能描述
MUL X,B,C	X←B×C
ADD X,A,X	X←A+X
SUB Y,D,E	Y←D-E
DIV X,X,Y	X←X÷Y

(a)使用三地址指令

指　令	功能描述
MOV X,B	X←B
MUL X,C	X←X×C
ADD X,A	X←X+A
MOV Y,D	Y←D
SUB Y,E	Y←Y-E
DIV X,Y	X←X÷Y

(b)使用二地址指令

指　令	功能描述
LOAD D	AC←D
SUB E	AC←AC- E
STORE Y	Y←AC
LOAD B	AC←B
MUL C	AC←AC×C
ADD A	AC←AC+A
DIV Y	AC←AC÷Y
STORE X	X←AC

(c)使用一地址指令

图 3-4　计算表达式 X=(A+B×C)÷(D−E)的程序

3.2.3　指令字长

指令字长取决于操作码的长度、操作数地址的长度和操作数地址的个数。不同机器的指令字长是不同的。一般来说，指令的设计应注意以下几点。

(1)由于机器字长、输入/输出数据的字长都是字节的整数倍，因此，指令字长也是字节的整数倍，如 8 位、16 位、32 位、48 位、64 位等。

(2)指令字中应划出足够的位数来表示机器的全部操作功能。例如，机器具有 50 种操作，则操作码至少需要 6 位。

(3)在指令采用定长结构的情况下，操作码和地址码两部分有一定的制约关系。例如，指令字长 16 位，操作码占 6 位，地址码就只有 10 位，可寻址的范围为 $2^{10}=1\ 024$ 个地址。要扩大寻址范围势必减少操作码的位数。为了解决这个矛盾，一般采用以下方法。

①地址码字段用寻址方式和寻址寄存器编号两部分表示，操作数的有效地址按寻址方式和指定寄存器的内容经计算后得出。由于寄存器的长度比地址码字段要短，因此可以有效地扩大寻址范围。

②突破固定字长指令的约束，采用可变字长指令结构。视指令功能、寻址范围、操作数的多少而采用单字节、双字节、3 字节或更多字节长度来表示一条指令。原则上讲，短指令比长指令好，主要因为短指令能节省存储空间、提高取指令的速度，但同时它也有很大的局限性。而长指令可能会占用 2 个或更多个地址，取指令的时间相对来说就要长一些，但它可以扩大寻址范围或带几个操作数。总之，短指令与长指令各有所长，具体采用哪种结构可根据实际要求而定。

3.3　指 令 类 型

一台计算机指令集中的所有指令按其功能可分为数据传送、数据运算、程序控制、输入/输出、数据转换等几种类型，如表 3-3 所示。

表 3-3　常见的指令类型

类　型	操作名	说　明
数据传送	MOVE	由源向目标传送字或块
	STORE	由处理器向存储器传送字
	LOAD	由存储器向处理器传送字
	EXCHANGE	源和目标交换内容
	CLEAR(RESET)	传送全 0 字到目标
	SET	传送全 1 字到目标
	PUSH	由源向栈顶传送字
	POP	由栈顶向目标传送字
数据运算	ADD	计算两个操作数的和
	SUBTRACT	计算两个操作数的差
	MULTIPLY	计算两个操作数的积
	DIVIDE	计算两个操作数的商
	ABSOLUTE	以其绝对值替代操作数
	NEGATE	改变操作数的符号
	INCREMENT	操作数加 1
	DECREMENT	操作数减 1
	AND	执行逻辑与操作
	OR	执行逻辑或操作
	NOT	执行逻辑非操作
	EXCLUSIVE-OR	执行逻辑异或操作
	TEST	测试指定的条件;根据结果设置标志
	COMPARE	对两个或多个操作数进行逻辑或算术比较;根据结果设置标志
	SET CONTROL VARIABLES	出于保护目的,中断管理或时间控制等进行设置控制的一类指令
	SHIFT	左(右)移位操作数,一端引入常数
	ROTATE	循环左(右)移位操作数
控制转移	JUMP(branch)	无条件转移;向 PC 装入指定地址
	JUMP condition	测试指定的条件,根据条件将指定地址装入 PC 或什么也不做
	JUMP sub	将当前程序控制信息放到一个已知位置;转移到指定地址
	RETURN	由已知位置内容替代 PC 和其他寄存器的内容

续表

类　　型	操 作 名	说　　明
控制转移	EXECUTE	由指定位置取操作数并作为指令执行；不修改 PC
	SKIP	PC 加 1 以跳过下一指令
	SKIP condition	测试指定条件；基于条件跳步或不跳步
	HALT	停止程序执行
	WAIT(hold)	暂停程序执行；重复测试指定条件；当条件满足时恢复执行
	NO operation	无操作完成，但程序执行继续
输入/输出(I/O)	INPUT	由指定的 I/O 端口或设备传送数据到目标(如主存或处理器寄存器等)
	OUTPUT	由指定的源传送数据到 I/O 端口或设备
	START I/O	向 I/O 处理器传送指令以初始化 I/O 操作
	TEST I/O	由 I/O 系统向指定目标传送状态信息
数据转换	TRANSLATE	根据一个对应表转换某一段内存的值
	CONVERT	将字的内容由一种形式转换到另一种形式(例如，将压缩的十进制数转换为二进制数等)
	MOVS	将原串传送到目的串
	CMPS	将原串与目的串逐个字节进行比较
	SCAS	在目的串中查找需要的字符

3.3.1　数据传送类指令

数据传送类指令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以下几种：

1)一般传送指令

一般传送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。传送通常以字节、字、双字或数组为单位，特殊情况下也可以位为单位进行传送，这类指令比较集中地体现了各种寻址方式。一般传送指令常用助记符 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。

(1)主存单元之间的传送。将主存源单元的内容传送到主存的另一目的单元，如 MOV M_2，M_1，其含义为

$$(M_2) \leftarrow (M_1)$$

注意：有些计算机不允许数据在两个主存单元之间直接进行传送，如 Intel x86 架构的计算机等。

(2)从主存单元传送到寄存器。将主存某一单元的内容传送到某一目的寄存器，即取数

操作，如 MOV R，M，其含义为

$$R \leftarrow (M)$$

在有些计算机中，该指令用助记符 LOAD 表示。

(3)从寄存器传送到主存单元。将源寄存器内容传送到主存某一目的单元，即存数操作，如 MOV M，R，其含义为

$$M \leftarrow (R)$$

在有些计算机中，该指令用助记符 STORE 表示。

(4)寄存器之间的传送。将源寄存器的内容传送到另一目的寄存器，如 MOV R_2，R_1，其含义为

$$R_2 \leftarrow (R_1)$$

2)数据交换指令

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)互换位置。例如，Intel 8086 中的 XCHG 指令就实现了这一功能，源操作数可以存放在通用寄存器或主存单元中，而目的操作数只允许存放在通用寄存器中。

3)堆栈操作指令

堆栈操作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，在程序中往往成对出现。

因为堆栈是主存的一个特定区域，所以对堆栈的操作也就是对存储器的操作。寄存器内容进栈指令(PUSH R)和主存单元内容进栈指令(PUSH M)分别相当于一般传送指令中的第 3 种和第 1 种；出栈送寄存器指令(POP R)和出栈送主存单元指令(POP M)分别相当于第 2 种和第 1 种。

3.3.2 数据运算类指令

数据运算类指令包括算术运算指令、逻辑运算指令和移位运算指令。

1)算术运算指令

算术运算指令主要用于进行定点和浮点运算，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。

不同的计算机对这类指令的支持有很大的差别。有些低档的微型机或小型机，由于以价格便宜为目的，硬件结构比较简单，因此支持的算术运算指令也较少，一般只设二进制加、减、比较等最简单的指令；而一些高档机除了最基本的运算指令外，还设置了乘、除指令，浮点运算指令及十进制运算指令。例如，Intel 8086 的指令集具有定点加、减、乘、除指令，可处理带符号或无符号的 8 位和 16 位定点整数以及十进制整数(压缩的十进制数和非压缩的十进制数)。

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。

比较指令是减法指令的一个特殊变化，仍是进行两数相减的运算，但结果不回送，即不

保留“差”。比较指令的功能在于不破坏原来的两个操作数，而仅设置相应的标志位，为紧跟在其后的条件转移指令提供操作的依据，从而决定程序的走向。

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，在进行高位字（字节）加、减运算时，应考虑低位字（字节）的进位或借位，因此，指令集中除了普通的加、减指令外一般都设置了带进位加指令和带借位减指令。

2）逻辑运算指令

一般计算机都具有与、或、非、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）中的某些位（1 位或多位）进行操作，常见的应用有如下几种。

（1）按位检测。利用与指令可以屏蔽数据字（字节）中的某些位，以便于对其他位进行测试，如判断其为 0 还是为 1。

（2）按位清除。利用与指令还可以清除目的操作数的某些位，从而使之变为 0。

（3）按位设置。利用或指令可以使目的操作数的某些位置 1，而其他位保持不变。

3）移位指令

移位指令分为算术移位指令、逻辑移位指令和循环移位指令三类，它们又可分为左移和右移两种，具体介绍如下。

（1）算术移位指令。算术移位指令的对象是带符号数，在多数机器中用补码表示。算术移位过程中必须保持操作数的符号不变，当左移一位时，数值增大一倍（不产生溢出）；而右移一位则数值减少一半（不考虑因移出舍去的末位尾数）。对于左移，空出的最低位将补 0；对于右移，空出的最高位将补符号位。

（2）逻辑移位指令。逻辑移位指令的对象是没有数值含义的二进制代码，因此移位时不必考虑符号问题。左移和右移的空出位都补 0。

（3）循环移位指令。循环移位按是否与进位一起循环又分为两种，即小循环（不带进位循环）和大循环（带进位循环）。

所有移位指令的操作数既可以在通用寄存器中，也可以在主存单元中，指令可以对操作数左移、右移 1 位或若干位。算术左移或右移 1 位，分别实现对带符号数乘以 2 或除以 2 的运算，对于没有乘、除法器的计算机，移位指令的这个性质显得特别重要；而逻辑左移或逻辑右移只是使操作数的数码位置发生变化，可以用来实现串并转换。

3.3.3　程序控制类指令

程序控制类指令用于控制程序的执行方向，并使程序具有测试、分析与判断能力。因此，它们是指令集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、程序自中断指令。

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方向。转移指令又分无条件转移指令和条件转移指令两种,如下所示。

(1)无条件转移指令。无条件转移指令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执行,其助记符一般为 JMP。

与无条件转移指令相似的还有一条跳步指令,它只跳过一条指令,其助记符一般为 SKIP。

(2)条件转移指令。条件转移必须受到条件的约束,若条件满足时才执行转移,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作出判断。转移的条件一般是指上次运算结果的某些特征或标志,主要包括进位标志(C)、结果为零标志(Z)、结果为负标志(N)、结果溢出标志(V)和结果奇偶标志(P)等,这些标志的组合可以产生 10 几种条件转移,如零转、非零转、正转、负转等。

无论是条件转移指令还是无条件转移指令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(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直接给出,即 A→PC。

2)子程序调用和返回指令

在编写程序时,通常将一些需要重复使用并能独立完成某种特定功能的程序段单独编成子程序,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。

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 等,这个过程又称为多重转子。另外,子程序还允许自己调用自己,即子程序递归调用。

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;而从子程序转向主程序的指令称为返回指令,其助记符一般为 RETURN。CALL 指令安排在主程序中需要调用子程序的地方,它是一地址指令。子程序的最后一条返回指令通常是零地址指令。转子指令和返回指令也可以是带条件的,条件转子和条件返回与前述条件转移的条件是相同的。

图 3-5(a)说明了如何使用子程序来构造程序。此例中有一个起始位置为 6 000 的主程序。这个主程序包括一条调用子程序 Sub1 的指令,Sub1 的起始位置为 6 500。当遇到这条调用指令时,CPU 挂起主程序的执行,并在 6 500 位置处取出下一条指令并开始执行 Sub1。在 Sub1 内,有两次对位于 6 800 位置的 Sub2 的调用,每次 Subl 都被挂起而执行 Sub2。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 3-5(b)说明了上述执行顺序。

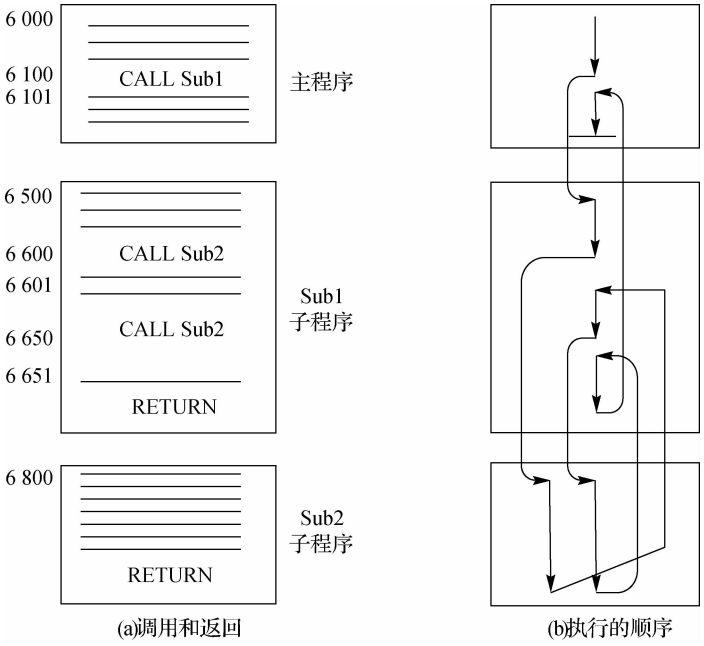


图 3-5　子程序嵌套

CPU 执行 CALL 指令时，除了要转到子程序的入口地址处，同时还需要保留返回地址，以便子程序能准确返回。保存返回地址的方法有以下几种。

(1)用子程序的第一个字单元存放返回地址。转子指令把返回地址存放在子程序的第一个字单元中，子程序从第二个字单元开始执行。返回时将第一个字单元地址作为间接地址，采用间址方式返回主程序。这种方法可以实现多重转子，但不能实现递归循环，Cyber 70 采用的就是这种方法。

(2)用寄存器存放返回地址。转子指令先把返回地址放到某一个寄存器中，再由子程序将寄存器中的内容转移到另一个安全的地方，如存储器的某个区域。这是一种较为安全的方法，可以实现子程序的递归循环。IBM 370 采用了这种方法，这种方法相对增加了子程序的复杂程度。

(3)用堆栈保存返回地址。无论是多重转子还是子程序递归，最后存放的返回地址总是最先被使用，堆栈的后进先出存取原则正好支持实现多重转子和递归循环，而且不增加子程序的复杂程度。这是应用最为广泛的方法，PDP-11、VAX-11、Intel 80x86 等均采用这种方法。

3)程序自中断指令

所谓中断是指计算机在运行程序的过程中出现急需处理的意外事件，必须暂停现在运行的程序去处理这些事件，处理结束后又返回到原程序继续运行。

中断通常是由计算机内部突发事件或外部设备的请求而随机产生的，但在有些计算机中，为了在程序调试中设置断点或实现系统调用功能，也设置了专门的自中断指令。由于这些指令是由软件驱动的，所以又称为软中断，如 Intel 80x86 中的中断指令 INT n。INT n 指令可暂停其后继指令的执行，转去执行相应的中断服务程序。但指令中不直接给出中断服

务程序入口地址，而只给出中断类型码 n，CPU 根据中断类型码可从中断向量表中找到中断服务程序的入口地址。有关中断问题的详细讨论将在第 7 章中进行。

无论中断是由硬件还是由软件驱动的，要退出中断过程都必须有一条中断返回指令(IRET)，中断返回指令总是安排在中断服务程序的出口处，迫使 CPU 返回到原来程序的断点处继续执行。

3.3.4 输入/输出类指令

输入/输出(I/O)类指令用来实现主机与外部设备之间的信息交换，包括输入/输出数据、主机向外设发送控制命令或外设向主机报告工作状态等。从广义角度来看，I/O 指令可以归入数据传送类，实际上有些计算机的 I/O 操作确实是由数据传送类指令实现的。

不同计算机的 I/O 指令差别很大，通常有两种方式，即独立编址方式和统一编址方式。

1)独立编址的 I/O

独立编址方式使用专门的输入/输出指令，以主机为基准，信息由外设传送到主机称为输入，反之称为输出。指令中应给出外部设备号或端口地址，这些设备号或端口地址与主存地址无关，是另一个独立的地址空间。

以 Intel 8086 为例，I/O 指令必须使用累加器 AX(以字为单位)或 AL(以字节为单位)进行传送，(见表 3-4)。在 I/O 指令中可以直接给出外设端口地址，该地址可以指定 0～255 范围内的任一端口；也可以由 DX 寄存器间接给出外设端口的地址，该地址可以指定 0～65 535 范围内的任一端口。

表 3-4 Intel 8086 的 I/O 指令

助 记 符	操 作 数	完成的操作
IN	Acc，Port	将指定端口中的内容输入 AL 或 AX 中
IN	Acc，DX	将 DX 寄存器所指定的端口中的内容输入 AL 或 AX 中
OUT	Port，Acc	将 AL 或 AX 的内容输出到指定的端口中
OUT	DX，Acc	将 AL 或 AX 的内容输出到由 DX 寄存器所指定的端口中

2)统一编址的 I/O

所谓统一编址就是把外设寄存器和主存单元统一编址。在这种方式下，不需要专门的 I/O 指令，而是使用一般的数据传送类指令来实现 I/O 操作。一个外设通常至少有两个寄存器，即数据寄存器和命令/状态寄存器，每个外设寄存器都可以由分配给它们的唯一的主存地址来识别，CPU 可以像访问主存一样去访问外设的寄存器。PDP-11 机采用的就是统一编址方式，它将最高的 4 KB 主存地址作为外设寄存器的地址。

这两种方式各有优缺点，如表 3-5 所示。

表 3-5　两种编址方式的比较

优 缺 点	独立编址方式	统一编址方式
优点	I/O 指令和访存指令容易区分； 外设地址线少，译码简单； 主存空间不会减少	总线结构简单； 全部访存类指令都可用于控制外设； 可直接对外设寄存器进行各种运算
缺点	控制线增加了 I/O Read 和 I/O Write 信号	占用主存一部分地址，缩小了可用的主存空间

3.3.5　数据转换类指令

在现代一些功能较强的计算机中，常设置有专门用于数据转换的指令。

1)数据转换指令

数据转换指令包括数制转换指令和数据类型转换指令。数制转换指令主要是将十进制数转换为二进制数，或反之；数据类型转换指令主要完成定点数与浮点数之间的转换。

2)字符串操作指令

字符串操作指令对提高机器处理数据的能力很有效，主要实现对字符串的传送、比较、查找等操作。一个字符串的参数有首地址和串长，因此，字符串操作指令都比较长，例如，字符串传送指令就需要 3 字节的地址字段，分别表示源字符串的首地址、目的字符串的首地址和串长。

3)压缩和扩展指令

压缩和扩展指令是指将一定长度的数据或字符变为另一长度的数据或字符的指令。将较短的数据格式转换成较长的数据格式称为扩展指令，一般采用符号扩展方法予以实现；而将较长的数据格式转换为较短的数据格式称为压缩指令，一般通过截去若干高位或按机器的某些约定来实现。

3.4　寻 址 方 式

寻址方式是指 CPU 确定本条指令的数据地址和下一条将要执行的指令地址的方式，它与计算机硬件结构紧密相关，而且对指令的格式和功能有很大影响。

3.4.1　有效地址的概念

由于指令长度的限制，指令中的地址码不会很长，而主存的容量却越来越大。以 IBM PC/XT 机为例，其主存容量可达 1 MB，而指令中的地址码字段最长只有 16 位，仅能访问 64 KB 空间。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，但为了灵活方便地编制程序，也需要对地址进行必要的变换。因此，将指令中地址码字段给出的地址称为形式地址，记为 A，形式地址并不代表操作数的真实地址，有可能不能直接用于访问主存。而对形式地址进行一定计算所得到的操作数真实地址称为有效地址，记为 EA(effective address)。

3.4.2 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。

指令寻址包括顺序寻址和跳跃寻址。顺序寻址只需通过程序计数器 PC 加 1(实际上是当前指令的字节数)即可自动形成下一条指令的地址；跳跃寻址则通过程序控制类指令来实现。跳跃寻址的转移地址形成方式有三种，即直接(绝对)寻址、相对寻址和间接寻址，它们分别与数据寻址中的直接寻址、相对寻址和间接寻址相类似，只不过其寻找到的是转移的有效地址而不是操作数的有效地址而已。

由于操作数可能包含在指令中、存放在主存单元或 CPU 的某一寄存器中，还可存放在堆栈或 I/O 设备的端口中，因此操作数的寻址方式较多。根据操作数存储位置的不同，可以派生各种不同的寻址方式，不同的计算机往往具有不同的寻址方式。在一些计算机中，还可以将某些寻址方式进行组合使用，从而形成更复杂的寻址方式。

3.4.3 基本的数据寻址方式

有的计算机寻址方式种类较少，在指令的操作码中就可以直接表示出寻址方式，而有的计算机采用多种寻址方式，此时在指令中专设一个字段来表示寻址方式。

下面介绍大多数计算机常用的几种基本数据寻址方式。

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，就取出了可以立即使用的操作数，因此，这样的操作数被称为立即数。其指令格式如下：

操作码 OP	操作数 A

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，不必再次访问存储器，提高了指令的执行速度。但是，因为操作数是指令的一部分，所以不能被修改；而且对于定长指令格式，操作数的大小将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，或者用于提供一个常数。

2)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 EA，即 EA＝A。由于这时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，因此又称其为绝对寻址方式，直接寻址过程如图 3-6 所示。

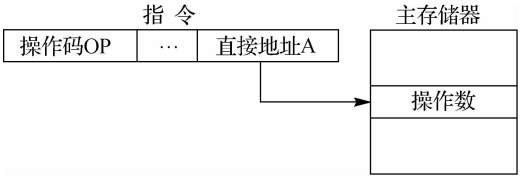


图 3-6　直接寻址过程

在早期的计算机中，主存的容量较小，指令中地址码的位数不长，采用直接寻址方式简单快速，也便于硬件实现，因此，常作为主要的寻址方式。但是，目前随着计算机主存容量的不断扩大，所需的地址码也越来越长。指令中地址码的位数不能满足整个主存空间寻址的要求，因此直接寻址方式受到了很大限制。另外，在指令执行过程中，为了取得操作数，必须进行访存操作，从而降低了指令的执行速度。

3)间接寻址

间接寻址意味着指令的地址码部分给出的地址 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。操作数有效地址的计算公式为：

$$EA=(A)$$

间接寻址过程如图 3-7 所示。

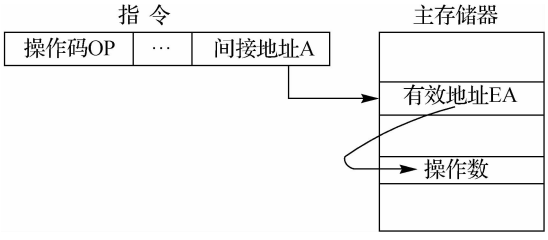


图 3-7　间接寻址过程

间接寻址的主要特点是：操作数的有效地址在主存储器中，可以被灵活修改而不必修改指令，从而比直接寻址灵活得多。但是，间接寻址在指令执行过程中至少需要两次访问主存才能取出操作数，从而严重降低了指令执行的速度。

4)寄存器直接寻址

寄存器直接寻址是指在指令的地址码部分给出 CPU 内部某一通用寄存器的编号，指令的操作数存放在相应的寄存器中，即 $EA=R_i$。其寻址过程如图 3-8 所示。

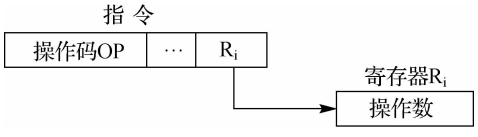


图 3-8　寄存器直接寻址过程

这种寻址方式具有如下两个明显的优点：

(1)由于寄存器在 CPU 的内部，因此指令在执行时从寄存器中取操作数比访问主存要快得多。

(2)由于寄存器的数量较少，因此寄存器编号所占位数也较少，从而可以有效减少指令

的地址码字段的长度。

因为寄存器直接寻址方式可以缩短指令长度，提高指令的执行速度，几乎所有计算机都使用了这种寻址方式。

5)寄存器间接寻址

为了克服间接寻址中多次访存的缺点，可采用寄存器间接寻址方式，即将操作数放在主存储器中，而操作数的地址放在某一通用寄存器中，然后在指令的地址码部分给出该通用寄存器的编号，这时有 EA＝(R_i)。其寻址过程如图 3-9 所示。

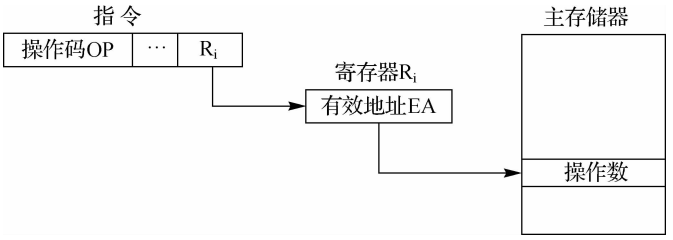


图 3-9　寄存器间接寻址过程

这种寻址方式的指令较短，并且在取指后只需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要快，也是目前计算机中使用较为广泛的一种寻址方式。

6)变址寻址

变址寻址是将指令的地址码部分给出的基准地址 A 与 CPU 内某特定的变址寄存器 R_x 中的内容相加，以形成操作数的有效地址，即 EA＝A＋(R_x)。使用哪一个寄存器作为变址寄存器必须在硬件设计时就事先规定好，变址寄存器 R_x 中的内容称为变址值，该值可正可负。其寻址过程如图 3-10 所示。

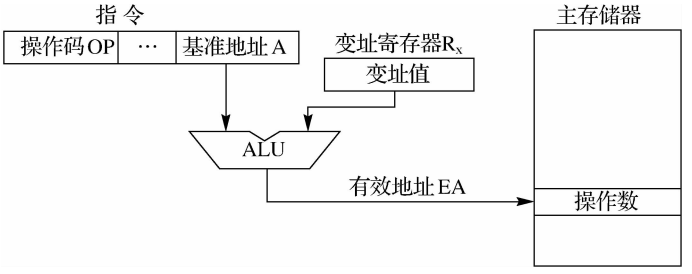


图 3-10　变址寻址过程

变址寻址方式是一种被广泛采用的寻址方式，最典型的应用就是将指令的地址码部分给出的地址 A 作为基准地址，而将变址寄存器 R_x 中的内容作为修改量。在遇到需要频繁修改操作数地址的情况时，无须修改指令，只要修改 R_x 中的变址值就可以了，这对于数组运算、字符串操作等一些进行成批数据处理的指令是很有用的。

7)基址寻址

基址寻址方式与变址寻址方式很相似，不同的是：在基址寻址方式中，指令的地址码部分给出偏移量 D，而基准地址存放在基址寄存器 R_b 中，最后操作数的有效地址仍然由基准地址 A 与偏移量 D 相加而成，即 EA＝(R_b)＋D。使用哪一个寄存器作为基址寄存器也必

须在硬件设计时就事先规定好，基址寄存器 R_b 中的内容称为基准地址，该值可正可负。其寻址过程如图 3-11 所示。

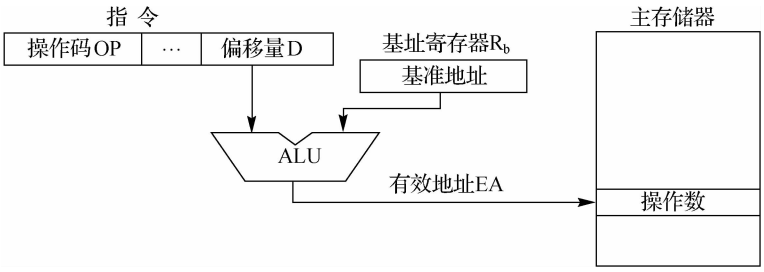


图 3-11　基址寻址过程

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，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，用来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，而在某些小型机和微型机中，基址寻址和变址寻址实际上已合二为一。

8）自相对寻址

自相对寻址也称程序计数器寻址（program counter addressing）。在这种寻址方式中，由程序计数器 PC 提供基准地址，而指令的地址码部分给出相对位移量 D，两者相加后作为操作数的有效地址，即 EA＝(PC)＋D。其寻址过程如图 3-12 所示。

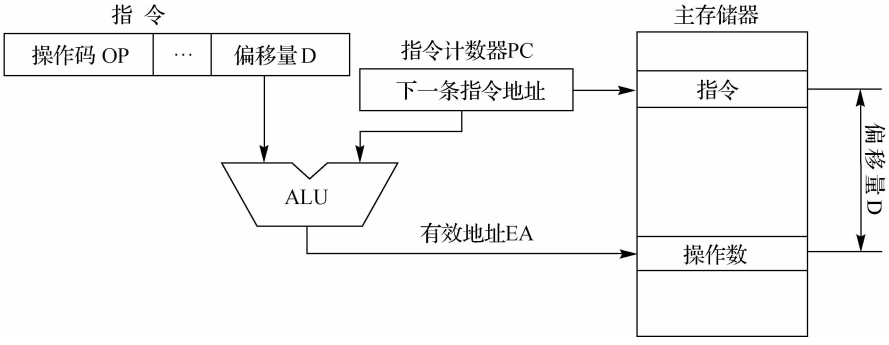


图 3-12　自相对寻址过程

自相对寻址方式使程序模块可采用浮动地址，编程时只要确定程序内部操作数与指令之间的相对距离即可，而无需确定操作数在主存中的绝对地址，这样，将程序安排在主存的任意位置都不会影响程序执行的正确性。

图 3-13 表明了每种寻址方式中地址是如何生成的。

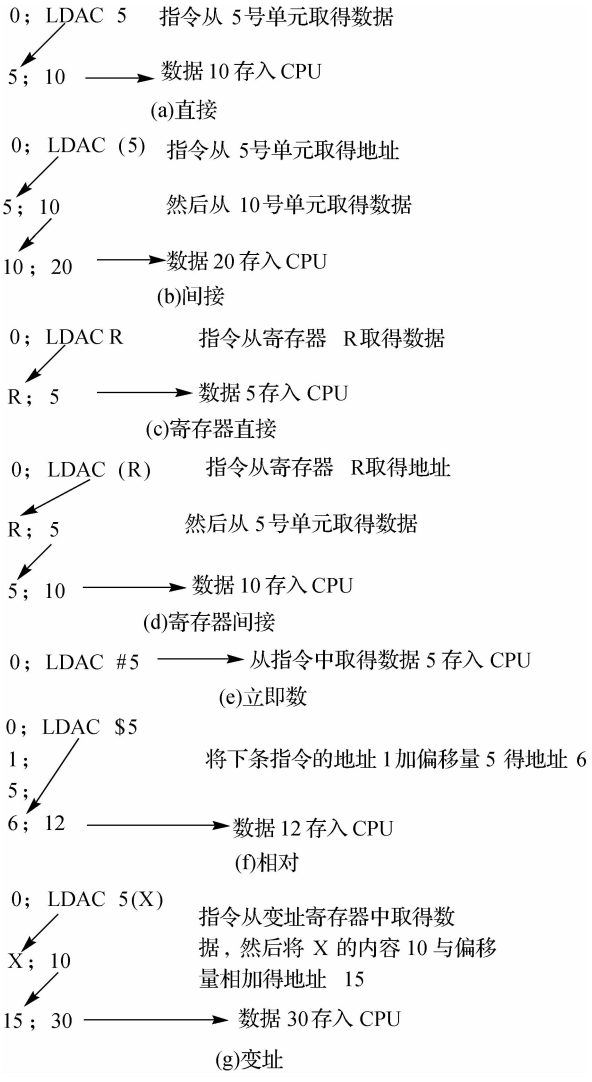


图 3-13 使用不同寻址方式产生地址

3.5 RISC 和 CISC 的基本概念

指令集结构的发展有两种截然不同的方向：一种是强化指令功能，实现软件功能向硬件功能的转移，基于这种指令集结构而设计实现的计算机系统称为复杂指令集计算机(CISC)；另一种是 20 世纪 80 年代发展起来的精简指令集计算机(RISC)，其目的是尽可能降低指令集结构的复杂性，以达到简化硬件实现、提高性能的目的。

3.5.1 推动 CISC 发展的目的

推动 CISC 发展的两个基本目的是改善性能和简化编译器。

在计算机发展过程中，指令集是伴随计算机体系结构的完善和性能的提高而发展起来

的。早期的计算机结构简单，指令集中包含的指令条数较少，功能也较弱，所以计算机的性能较差。如 20 世纪 70 年代的小型机和一些低档的 8 位计算机，其基本指令一般仅有几十条。从指令集的完备性和有效性的角度来看，当然希望指令集更丰富、功能更强。随着 VLSI 技术的迅速发展，硬件成本不断下降，软件成本不断上升，促使人们在指令集中增加更多和更复杂的指令，以适应不同应用领域的需要。特别是系列机问世之后，为了达到软件兼容的目的，新设计的机型或高档机除了要继承老机器指令集中的全部指令外，还要增加若干新指令，从而导致同一系列计算机的指令集越来越复杂，机器结构也越来越复杂。目前，大多数计算机的指令集包含多达几百条指令，例如，VAX-11/780 机有 303 条指令，18 种寻址方式。这体现了计算机性能越高，其指令集应越复杂的传统设计思想。

除了改善性能外，推动 CISC 的另一个目的是简化编译器。编译器的任务是将每个高级语言语句翻译成一个机器指令序列，若有类似于高级语言语句的机器指令，那么该任务就简单多了，因此人们采用更复杂的指令来试图设计能对高级语言提供更好支持的机器。

3.5.2　RISC 的产生

一般地，指令集的指令数目越多，CPU 的延时就越大。例如，若某 CPU 具有 4 位定长操作码，则需要使用一个 4-16 译码器来产生输出，以便对应指令集的 16 条指令；若 CPU 有 32 条指令，则需要一个 5-32 译码器；已知 5-32 译码器比 4-16 译码器需要更多的时间产生输出，因此导致 CPU 的最大时钟频率降低。

于是，一些设计者考虑从指令集中去掉一些使用频率较低的指令。他们认为在 CPU 中减小延时可以使 CPU 以高频率运行，从而更快地执行每条指令。然而，去掉的指令通常对应的是高级语言中的特殊语句，从指令集中去掉这些指令将迫使 CPU 用几条指令来替代完成同样功能的一条指令，这肯定会需要更多时间。根据去除指令的使用频率以及实现它们的功能所需的替代指令的条数，此方法不一定能提高系统性能。

考虑一个时钟周期为 20 ns 的 CPU，可以从指令集中去除一些指令，将时钟周期减至 18 ns。这些指令包含了一个典型程序中代码量的 2%，并且每一条指令必须被其他三条指令来替代。假设每条指令的执行需要相同的时钟周期数 C。如果这些指令没有从指令集中除去，那么所有的指令(100%)需要(20C) ns 来处理。如果它们被除去，那么剩余 98%的程序代码将需要(18C) ns，其余 2%的代码需要三倍的指令数，即(18C×3) ns＝(54C) ns。比较两者推出以下结论：

$$(100\% \times 20C) \text{ vs. } (98\% \times 18C + 2\% \times 54C)$$

$$20C \text{ vs. } (17.64C + 1.08C)$$

$$20 > 18.72$$

平均起来看，本例中指令较少的 CPU 的性能相对好些。但是，如果去除的指令数达到了典型程序的 10%，那么去除这些指令将降低整个 CPU 的性能。

通过对传统的 CISC 指令集进行测试发现，各种指令的使用频率相差很大。最常使用的是一些比较简单的指令，这类指令仅占指令总数的 20%，但在各种程序中出现的频率却占 80%，其余大多数指令是功能复杂的指令，这类指令占指令总数的 80%，但其使用频率很低，仅占 20%。因此，人们把这种情况称为“20%－80%定律”。从这一事实出发，人们开始了对

指令集合理性的研究，1975 年，IBM 公司的 John Cocke 提出了精简指令集的想法，之后，各种精简指令集计算机也随之诞生。

大部分 RISC 具有以下一些特点：

(1)选取使用频率最高的一些简单指令，以及一些很有用但不复杂的指令。

(2)指令长度固定，指令格式种类少，寻址方式种类少。

(3)只有取数/存数指令访问存储器，其余指令的操作都在寄存器之间进行。

(4)CPU 中通用寄存器数量相当多，远远超过传统 CISC 的 CPU 中寄存器的数量。

(5)大部分指令都能在一个机器周期内完成，实际上是指在采用流水线结构以后，每个机器周期内能完成一条指令功能，而并不是说一条指令从取指到完成指定功能只要一个机器周期。

(6)以硬布线控制为主，不用或少用微指令码控制。

(7)一般用高级语言编程，特别重视编译优化工作，以减少程序执行时间。

3.5.3 CISC 与 RISC 的特征对比

表 3-6 比较了几个典型的 RISC 和 CISC 系统，各有优点所在。

表 3-6 典型 CISC 和 RISC 的特征

比较项目	CICS			RISC		
	IBM 370/168	VAX 11/780	Intel 80486	SPAR C	MIPSR 4000	PowerPC
开发年份	1973	1978	1989	1978	1991	1993
指令数量(条)	208	303	235	69	94	225
指令长度(B)	2～6	2～57	1～11	4	4	4
寻址方式	4	22	11	1	1	2
通用寄存器数(个)	16	16	8	40～520	32	32
控制存储器大小(KB)	420	480	246	—	—	—
cache 大小(KB)	64	64	8	32	128	16～32

RISC 的指令比 CISC 的指令更少、更简单，所以其控制单元不太复杂，比较容易设计。而且 RISC 处理器比 CISC 处理器可以运行更高的时钟频率，同时节省了处理器芯片上的大量空间，因此设计者可以利用这些空间设置附加的寄存器和其他组件。简单的控制单元也减少了开发成本。而且，具有更简单的设计，可以很容易地向 RISC CPU 的控制单元加入并行技术。

由于 RISC 指令集的指令较少，因此它的编译器比 CISC 处理器的编译器要简单些。作为一个总的指导方针，CISC 处理器最初是为汇编语言编程设计的，而 RISC 处理器是针对编译器和高级语言编程设计的。但是，编译同一个高级语言程序，RISC CPU 将需要比 CISC CPU 更多的指令。

CISC 方法也提供了一些优点。尽管 CISC 处理器更加复杂，但是这些复杂性不需要增加开发成本。目前的 CISC 处理器经常是某个完整处理器家族的最新近补充，如 Intel Pentium 家族。同样地，它们可以组合其家族中早期处理器的部分设计，从而减少了设计成本，提高了可靠性。

CISC 处理器还提供了与其家族其他处理器的向后兼容性。有了向后兼容性，如果它们引脚兼容，那么可以很简单地用最新产品替代前代处理器，而不用改变计算机设计的其余部分。同时，无论引脚兼容与否，都允许 CISC CPU 运行其家族先前处理器使用的相同软件。例如，一个程序成功运行在 Pentium Ⅱ上，那么它应该也能在 Pentium Ⅲ上成功运行。这为用户节省了巨大的开销，而且可以确定一个微处理器的成功或失败。

目前处理器设计的主流是 RISC 和 CISC 的相互融合，人们逐渐认识到：在 RISC 设计中包括某些 CISC 特色会有好处；而在 CISC 设计中包括某些 RISC 特色也会有益。新型的处理器家族，例如，PowerPC 处理器吸取了 RISC 方法和 CISC 方法的优点，开发出了 RISC 和 CISC 的混合产品。

3.6　实例：Intel x86 和 ARM 的指令集结构

目前的计算机有两大典型系列，即 Intel x86 系列和 ARM（先进的 RISC 机器）系列。它们共同概括了当前计算机设计的趋势。Intel x86 结构基本上是一个带有某些 RISC 特征的复杂指令集计算机，而 ARM 本质上是一个精简指令集计算机。

3.6.1　Intel x86 的指令集结构

下面分别介绍 Intel x86 的指令格式、数据类型、指令类型和寻址方式。

1）指令格式

Intel x86 包含各种指令格式。在各指令元素中，操作码字段是必需的，其他都是可选的。图 3-14 说明了 x86 的指令格式。指令由 0～4 个可选的指令前缀、1 B 或 2 B 的操作码、一个由 Mod R/M 和 SIB（比例变址）字段组成的可选的地址指定符、一个可选的偏移量以及一个可选的立即数字段组成。

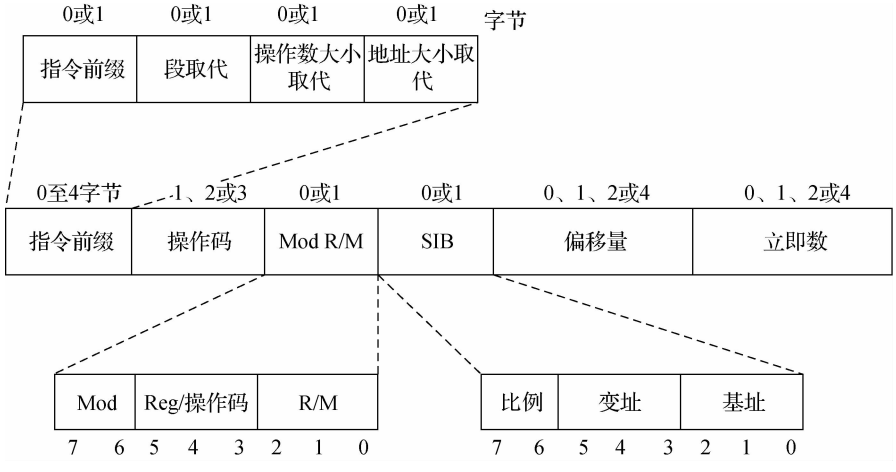


图 3-14　Intel x86 的指令格式

前缀字段包含如下4部分。

(1)指令前缀。若出现指令前缀,则它由LOCK(锁定)前缀或一个重复前缀组成。LOCK前缀用于在多处理器环境中保证对共享存储器的排他性访问。重复前缀指定串的重复操作,这就使得Intel x86处理串要比普通的软件循环快得多。有5种重复前缀,包括REP、REPE、REPZ、REPNE和REPNZ。当无条件的REP前缀出现时,将对串中连续元素重复执行指令中指定的操作,重复的次数由CX寄存器指定。有条件的REP前缀会引起指令重复执行,直到CX变为0或指定的条件被满足为止。

(2)段取代。段取代显式地指定这条指令应使用哪个段寄存器,取代x86为该指令生成的默认段寄存器。

(3)操作数大小。操作数大小指令默认的操作数是16位还是32位,操作数大小前缀用于在16位和32位的操作数之间进行切换。

(4)地址大小。地址大小处理器能使用16位或32位地址来寻址存储器。地址大小确定了指令格式中偏移量的大小和在有效地址计算中生成的位移量大小,其中的一种被设计成默认值。地址大小前缀还用于32位和16位地址生成之间的切换。

Intel x86指令本身包括如下字段:

(1)操作码。操作码字段为1 B、2 B或3 B长度。操作码除了指明操作性质外,还可指定数据是字节还是全尺寸(16位或32位,取决于上下文)、数据操作方向(送至或来自存储器)以及立即数字段是否需要进行符号扩展等信息。

(2)Mod R/M。Mod R/M这个字节和下一个字节提供寻址信息。Mod R/M字节指定操作数是在寄存器中还是在存储器中。若在存储器中,则该字节中的一个字段指定将要使用的寻址方式。Mod R/M字节由三个字段组成:Mod字段(2位),它与R/M字段组合构成32个可能的值,即8个寄存器和24个变址方式;Reg/操作码字段(3位)指定一个寄存器号或者用做操作码信息的3个补充位;R/M字段(3位)能指定一个寄存器作为一个操作数的位置,或者它构成寻址方式的一部分,与Mod字段组合来编码。

(3)SIB。Mod R/M字节的某些编码要求包含SIB字节来完整地确定寻址方式。SIB字节由三个字段组成:比例字段(2位)指定用于比例变址的比例因子;变址字段(3位)用于指定变址寄存器;基址字段(3位)用于指定基址寄存器。

(4)偏移量。当地址方式指定符指出需要使用偏移量时,一个8位、16位或32位有符号整数的偏移量被加入指令中。

(5)立即数。立即数指在指令中提供的一个8位、16位或32位的操作数值。

由上可见,在Intel x86格式中,寻址方式是作为操作码序列的一部分来提供的,而不是与每个操作数一起提供。因为只能一个操作数有寻址方式信息,所以,Intel x86指令中也就只能引用一个存储器操作数。因此,Intel x86的指令紧凑,但若要求存储器到存储器的操作,Intel x86需要使用多条指令才能实现。

Intel x86指令格式允许变址使用1 B、2 B或4 B位移,虽然使用较长的变址位移会导致指令更长,但这个特点能提供所需的灵活性。例如,在寻址大的数组或大的栈帧时它就很有用。

Intel x86指令集的编码是很复杂的。一方面是由于8086向下兼容的需要;另一方面是由于设计者打算为编译器设计者提供尽可能多的支持,以产生更有效的代码。然而,设计这

样复杂的指令集还是使用 RISC 指令集是一个有争议的事情。

2)数据类型

Intel x86 能处理 8 位(字节),16 位(字),32 位(双字),64 位(四字)和 128 位(双四字)等各种长度的数据类型。为了允许最大的数据结构灵活性和最有效地使用存储器,不需要遵循数据对齐规则,即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 倍整数地址上对齐,其他类推。然而,当由 32 位总线存取数据时,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。Intel x86 采用小端序(little-endian ordering)风格,即最低有效字节存于最低地址中。

字节、字、双字、四字和双四字称为常规数据类型。另外,Intel x86 支持一系列的特殊数据类型,如未压缩的 BCD、压缩的 BCD、近指针、远指针、位字段、字节串、浮点数、压缩的 SIMD(single-instruction-multiple-data,单指令多数据)等,这些特殊数据类型只被特殊指令所识别和操作。

在 Intel x86 中,带符号整数采用 2 的补码表示,可以是 16 位、32 位或 64 位长。浮点数实际上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,有三种浮点数表示,均符合 IEEE 754 标准。压缩的 SIMD 数据类型是为了优化多媒体应用的性能而作为一种对指令集的扩展添加到 Intel 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),基本的思想是把多个操作数打包为一个内存引用项,并且并行地操作这些操作数,从而提高性能。

3)指令类型

Intel x86 的大多数指令是在其他机器指令集中也能找到的常规指令,但还提供了一些专为 Intel x86 体系结构精心设计的特殊指令,其目的是为编译程序编写人员提供一种强有力的工具,以将高级语言程序转换成优化的机器语言程序。表 3-7 列出了基本的指令类型。

表 3-7　Intel x86 基本的指令类型

指　令		说　明
数据传送	MOV	在寄存器之间或寄存器与主存之间传送操作数
	PUSH	将操作数压入堆栈
	PUSHA	将所有寄存器的内容压入堆栈
	MOVSX	传送字节、字、双字;符号被扩展。将字节传送到字或将字传送到双字,以 2 的补码符号扩展方式进行
	LEA	装入有效地址。将源操作数的偏移量装入目标操作数
	XLAT	查找翻译。以用户编制的翻译表的一字节替代 AL 中的字节。当 XLAT 执行时,AL 中应有表的无符号数的索引值。XLAT 以被索引项的内容替代 AL 的内容
	IN/OUT	由 I/O 端口输入,或输出到 I/O 端口

续表

指令		说明
算术	ADD	加操作数
	SUB	减操作数
	MUL	无符号整数乘法，操作数是字节、字或双字，结果是字、双字或四字
	IDIV	有符号数除法
逻辑	AND	逻辑与操作数
	BTS	位测试和置位，对位字段(域)操作数进行操作。指令将一位当前值复制到 CF 标志，并设置原来位为 1
	BSF	位向前扫描。扫描一个字或双字，当发现第一个 1 时，将该位对应的位号保存到一个寄存器中
	SHL/SHR	左或右的逻辑移位
	SAL/SAR	左或右的算术移位
	ROL/ROR	左或右的循环移位
	SETcc	根据状态标志定义的 16 个条件设置一字节为 0 或 1
控制传递	JMP	无条件转移
	CALL	转移控制到另一位置。在转移之前，将此 CALL 指令之后的指令地址压入栈
	JE/JZ	若相等/为 0，则跳转
	LOOPE /LOOPZ	若相等/为 0，则循环。这是一个使用 ECX 值的条件跳转指令。指令先减量 ECX，然后为转移条件测试 ECX 值
	INT/INTO	中断/上溢中断。转移控制到一个中断服务子程序
字符串操作	MOVS	传送字节、字或双字的字符串。它对由 ESI 和 EDI 索引的字符串元素进行操作，每次字符串操作之后，这两个寄存器自动递增或递减以指向下一个字符串元素
	LODS	装入字节、字或双字的字符串
高级语言支持	ENTER	创建一个能用来实现块结构化(block-structured)高级语言规则的栈帧
	LEAVE	上述 ENTER 动作的逆动作
	BPIMD	检查数组边界。验证第一个操作数是否在一个上下限范围之内，此上下限值存于被第二个操作数所引用的两个相邻存储器中。若值不在边界内，则产生中断。此指令用来检查数组索引是否越界
标志控制	STC	置位进位标志
	LAHF	将标志装入 A 寄存器。复制 SF、ZF、AF、PF 和 CF 位到 A 寄存器
段寄存器	LDS	装入指针到数据段寄存器和另一个寄存器

续表

指　令		说　明
系统控制	HLT	暂停
	LOCK	对共享存储器提出保持要求，这样在立即跟随 LOCK 之后的那条指令期间，Pentium 可排他性地使用共享存储器
	ESC	处理器扩展换码。此换码指示后面的指令，将在支持高精度整数和浮点数计算的数值协处理器上执行
	WAIT	等待，直到 BUSY＃信号撤除。挂起 Pentium 执行的程序，直到处理器已测出 BUSY 引脚信号无效，这表明数值协处理器已执行结束
保护	SGDT	保存全局描述符表
	LSL	装入段限。将段限装入一个用户指定的寄存器中
	VERR/ VERW	验证段是否可被读/写
高速缓存管理	INVD	清空内部高速缓存(cache)
	WBINVD	在将已修改的行写回存储器之后，清空内部 cache
	INVLPG	使一个页表高速缓存(translation lookaside buffer，TLB)项无效

4)寻址方式

Intel x86 内部有 6 个段寄存器，即 CS、DS、ES、SS、FS 和 GS，每一个段寄存器都有一个段描述符寄存器与之关联，段描述符寄存器对用户(包括机器语言程序员)而言是不可见的，它登记了段的起始地址、段的长度和段的访问权限。此外，还有两个寄存器，即基址寄存器和变址寄存器，用于构造地址。其地址转换过程如图 3-15 所示。具体使用哪一个段描述符寄存器来形成线性地址，可以根据指令和上下文信息来确定。

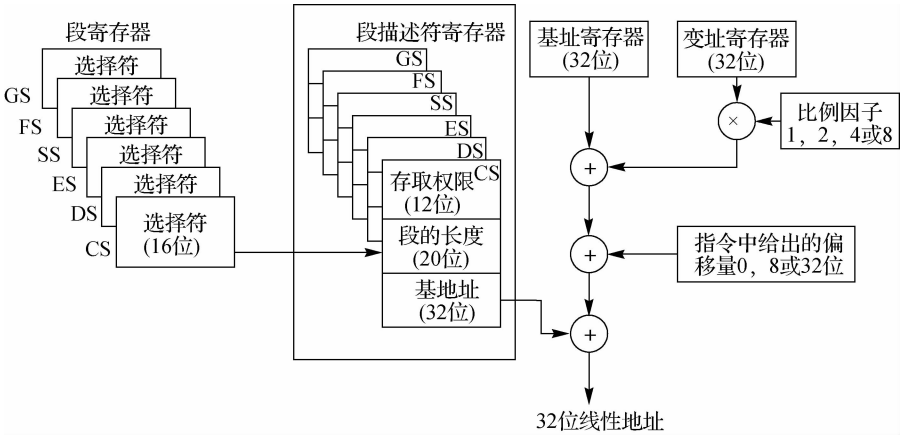


图 3-15　Intel x86 寻址方式的计算

如果不采用分页机制，那么 32 位线性地址就是操作数在主存储器中的物理地址；若采用分页机制，线性地址是由页目录、页号和页内偏移量三个部分构成的，则还必须通过一个页转换机制才能生成一个真正的 32 位物理地址。

Intel x86 具有丰富的寻址方式，如表 3-8 所示。

表 3-8　Intel x86 寻址方式

方　式	算　法
立即寻址	操作数＝A
寄存器寻址	LA＝R
偏移量寻址	LA＝(SR)＋A
基址寻址	LA＝(SR)＋(B)
基址带偏移量寻址	LA＝(SR)＋(B)＋A
比例变址带偏移量寻址	LA＝(SR)＋(I)×S＋A
基址带变址和偏移量寻址	LA＝(SR)＋(B)＋(I)＋A
基址带比例变址和偏移寻址	LA＝(SR)＋(I)×S ＋(B)＋A
相对寻址	LA＝(PC)＋A

注：LA＝线性地址，(X)＝X 的内容，SR＝段寄存器，PC＝程序计数器，A＝指令中地址字段的内容，R＝寄存器，B＝基址寄存器，I＝变址寄存器，S＝比例因子

对于立即数寻址方式，操作数被包括在指令中。操作数可以是字节、字或双字长的数据。

对于寄存器寻址方式，操作数位于某一通用寄存器中。对于数据传送、算术和逻辑运算等指令来说，操作数可以位于 32 位的通用寄存器(EAX、EBX、ECX、EDX、ESI、EDI、ESP 或 EBP)，也可以位于 16 位的通用寄存器(AX、BX、CX、DX、SI、DI、SP 或 BP)，还可以位于 8 位的通用寄存器(AH、BH、CH、DH、AL、BL、CL 或 DL)。对于浮点操作指令，要使用两个 32 位寄存器来构成一个 64 位操作数。另外，还有一些访问段寄存器(CS、DS、ES、SS、FS 或 GS)的指令。

对其他寻址方式来说，操作数都存放在主存储器中。

在偏移量方式中，指令中的偏移量可以为 8 位、16 位或 32 位，该偏移量反映了操作数距离段起点的位移。某些情况下，段是显式指定的；在另外一些情况下，段通过一个简单的段默认指派规则来隐式指定。

基址方式在指令中指定一个 8 位、16 位或 32 位 7684 寄存器，并且在该寄存器中含有操作数的有效地址。这就是前面讨论过的寄存器间接寻址。

在基址带偏移量方式中，指令中包括一个偏移量并指定基址寄存器，基址寄存器可以是任意一个通用寄存器。最后的操作数有效地址由基址寄存器的内容与偏移量相加

得到。

在比例变址带偏移量方式中，指令中除给出变址寄存器的编号和相应的偏移量外，还要给出比例因子。除 ESP 通常用于堆栈处理之外，其他任何通用寄存器都可以作为变址寄存器。计算操作数的有效地址时，变址寄存器的内容乘以比例因子（1、2、4 或 8），然后再加上偏移量即可。对于索引一个阵列，这种方式是很方便的。比例因子为 2 能用于一个 16 位整数阵列，比例因子为 4 能用于 32 位整数或浮点数，比例因子为 8 能用于一个双精度浮点数阵列。

基址带比例变址和偏移量方式是将变址寄存器内容乘以比例因子再加上基址寄存器的内容和指令中给出的偏移量，最后得到操作数的有效地址 EA。若一个阵列存于栈帧中，这种寻址方式是很有用的，此时阵列元素可以是 2 B、4 B 或 8 B 长。这种方式亦能对阵列元素是 2 B、4 B 或 8 B 长的二维阵列提供有效的索引。

相对寻址主要用于控制转移指令。

3.6.2　ARM 的指令集结构

ARM 的指令集结构包括指令格式、数据类型、指令类型和寻址方式。

1）ARM 指令格式

ARM 的所有指令都是 32 位的，并有规整的格式，如图 3-16 所示。指令的前 4 位是条件码，ARM 的所有指令都是条件执行的。接下来的 3 位指定了指令的一般类型。除分支指令外，其余大多数指令接下来的 5 位构成了操作码和/或操作的修订码。剩下的 20 位用于操作数寻址。ARM 指令的这种规整格式使得指令译码单元的工作变得比较轻松。

S = 对于数据处理指令，用于标示指令将更新条件码

S = 对于多装/多存指令，用于标示指令是否仅在特权模式才允许执行

P, U, W = 用于区分不同的寻址方式

B = 用于区分无符号的字节访问（B=1）或字访问（B=0）

L = 对于取数/保存指令，区分取数（L=1）和存数（L=0）

L = 对于分支指令，确定返回地址是否保存在连接寄存器中

图 3-16　ARM 指令格式

(1)条件码。ARM 体系结构定义了 4 个条件标志:N(负标志)、Z(零标志)、C(进位标志)和 V(溢出标志)。这些标志保存在程序状态寄存器 PSW 中,其值构成了 ARM 处理器的条件码。表 3-9 显示了条件执行所依赖的条件组合。ARM 中使用条件码有以下两个不同寻常之处。

①所有指令都有条件码字段。这意味着所有指令都可以根据条件决定是否执行。实际上,除了 1110 和 1111 这两种条件码以外,其他任何指令的条件码字段都意味着该指令仅当条件满足时才能执行。

②所有的数据处理指令都包含一个 S 位,用于指出该指令是否会修改条件标志。

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

表 3-9 ARM 指令条件执行的条件码

条件码	符号	被测试的条件标志	说明
0000	EQ	Z=1	相等
0001	NE	Z=0	不相等
0010	CS/HS	C=1	进位设置/无符号大于或等于
0011	CC/LO	C=0	进位清除/无符号小于
0100	MI	N=1	正/负
0101	PL	N=0	正/零
0110	VS	V=1	溢出
0111	VC	V=0	无溢出
1000	HI	C=1 AND Z=0	无符号大于
1001	LS	C=0 OR Z=1	无符号小于或等于
1010	GE	N=V [(N=1 AND V=1) OR (N=0 AND V=0)]	有符号大于或等于
1011	LT	N=V [(N=1 AND V=0) OR (N=0 AND V=1)]	有符号小于
1100	GT	(Z=0) AND (N=V)	有符号大于
1101	LE	(Z=1) OR (N≠V)	有符号小于或等于
1110	AL	—	总是(无条件)
1111	—	—	该指令只能无条件执行

(2)压缩指令集。压缩指令集是 ARM 指令集中一个重新编码的子集。设计压缩指令集的目的是提高使用 16 位或更窄内存数据总线的 ARM 性能,使其相对于普通 ARM 指令集来说有更高的代码密度。压缩指令集包含了 ARM 的 32 位指令集的子集,并重新编码为 16 位指令。下面列出了压缩指令集采取的精简措施。

①压缩指令都是无条件的,因此条件码字段都被省去。而且,所有的压缩算术和逻辑指令都更新条件标志,因此标志更新位也被省去。这样总共省去了 5 位。

②压缩指令只包含了全部指令集中的一部分操作,只用到 2 位的操作码字段,加上一个 3 位的类型字段。这样又省去了 2 位。

③接下来通过对操作数字段的精简,又省去了 9 位,从而总共省去了 16 位。例如,压缩指令只引用寄存器 r0～r7,因此只需要 3 位寄存器引用字段;立即数字段中也省去了 4 位循环移位量字段。

ARM 处理器可以执行压缩指令和普通 32 位 ARM 指令混合在一起的程序。处理器控制寄存器中的一位用于确定当前要运行的指令是哪种类型的指令。

2)ARM 数据类型

ARM 能处理 8 位(字节)、16 位(半字)和 32 位(字)数据。通常,对于数据访问遵循对齐规则;而对于非对齐数据的访问,则需进行特殊处理。

所有数据类型(字节、半字、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,即非负整数。所有数据类型也支持使用 2 的补码表示带符号整数。

大部分 ARM 处理器都不支持浮点硬件,这样可以节省功耗和芯片面积。如果某些处理器需要浮点运算,那么只能通过软件来提供。ARM 也可以带一个浮点协处理器来支持 IEEE 754 标准所规定的单精度和双精度浮点数据类型。

3)ARM 指令类型

ARM 提供了大量指令类型,主要类型如下。

(1)装载和保存指令。在 ARM 体系结构中,只有装载(load)和保存(store)指令能访问内存,运算类指令只对寄存器和指令中的立即数进行操作。ARM 体系结构支持两大类装载和保存指令:一类是装载和保存单个寄存器数据到内存;另一类是装载和保存一对寄存器的数据到内存。

(2)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 32 MB。由于程序计数器使用了通用寄存器 R15,因此直接改写 R15 的值也可以产生分支或跳转。子程序调用是通过修改标准的分支指令来实现的。因为可以向前或向后跳转 32 MB 的距离,所以分支链接(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)寄存器中。分支的确定是通过指令中的 4 位条件码字段完成的。

(3)数据处理指令。数据处理指令包括逻辑指令、加法和减法指令以及测试和比较指令。

(4)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 位操作数为输入,产生 64 位结果的乘法指令等。

(5)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，其中指令的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，主要应用于图像处理。

(6)扩展指令。扩展指令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。

(7)状态寄存器存取指令。状态寄存器存取指令是用于读/写状态寄存器中特定位的指令。

4)ARM 寻址方式

通常，RISC 都采用简单、直接的一组寻址方式，但 ARM 却有相对比较丰富的寻址方式，这些寻址方式基本是根据指令类型区分的。

(1)装载/保存寻址。ARM 中只有装载/保存指令能访问内存。内存地址通常由一个基址寄存器加上一个偏移量来得到，考虑到变址的情况，则可分为以下 3 种不同方式。

①偏移(offset)。对于这种寻址模式，变址不被使用。内存地址通过基址寄存器的值加上或减去偏移量而得到。

②前变址(preindexing)。内存地址的计算与偏移寻址的方式一样，计算得到的内存地址同样被写回基址寄存器。也就是说，基址寄存器增加或减少了一个偏移量的值。

③后变址(postindexing)。内存地址就是基址寄存器的值。之后，偏移量被加到基址寄存器值中或从寄存器值中减去，结果再保存回基址寄存器。

可以看到，ARM 中的基址寄存器在前变址和后变址寻址方式时，更像一个变址寄存器。偏移量可以是一个存于指令中的立即数，也可以是另一个寄存器的值。如果偏移量在寄存器中，那么就可以获得一个有用的特性，即带比例的寄存器寻址。在偏移量寄存器中保存的值可以通过移位操作按比例放大或缩小。移位操作可以是逻辑左移、逻辑右移、算术右移、循环右移以及扩展的循环右移。移位的位数可以通过指令中的立即值给出。

(2)数据处理指令的寻址。数据处理指令使用寄存器寻址或者寄存器和立即数混合寻址。采用寄存器寻址时，操作数寄存器中提供的值可以采用 5 种移位操作来进行按比例放大和缩小。

(3)分支指令的寻址。分支指令只有一种寻址方式，即立即寻址。分支指令中带有一个 24 位立即数，在计算地址时，这个立即数被左移 2 位，从而使地址对齐到 32 位字的边界。这样，24 位立即数可以产生相对于程序计数器来说 32 MB 范围内的有效地址。

(4)多装/多存寻址。多装(load multiple)指令从内存装载多个数到多个(可能全部)寄存器。多存(store multiple)指令把多个(可能全部)寄存器的内容保存到内存中。被装载或保存的寄存器列表由指令中一个 16 位的字段给出，其中每一位对应一个 16 位寄存器。多装和多存指令寻址模式会产生一组连续的内存地址。编号最小的寄存器对应最低的内存地址，编号最大的寄存器对应最高的内存地址。内存地址的产生有 4 种方式，即后递增(increment after)、先递增(increment before)、后递减(decrement after)以及先递减(decrement before)。指令用一个基址寄存器指定一个内存地址，寄存器的值从这个地址装载，或向这个地址存入，方向可以按字位置上升(递增)或下降(递减)。递增或递减可以发生在第一个值装载或保存完之后或之前。这些指令对于块装载和保存、栈操作以及过程退出操作是很有用的。

习　题　3

(1)简述指令集结构包含的 5 个重要因素。

(2)一条指令应包含哪些主要信息?

(3)在机器指令集内,典型的操作数类型是什么?

(4)控制转移指令的作用是什么?

(5)列出并简要说明生成条件的两种普通方式。

(6)列出为子程序返回指令保存返回地址的 3 种可能位置。

(7)分别以零地址、一地址、二地址和三地址法编写程序来计算下式:

$$X=(A+B\times C)/(D\times E-F)$$

并据此对 4 种地址机制进行比较。4 种地址机制可使用的指令分别如下表所示:

零地址	一地址	二地址	三地址
PUSH M	LOAD M	MOV (X←Y)	MOVE (X←Y)
POP M	STPORE M	ADD (X←X+Y)	ADD (X←Y+Z)
ADD	ADD M	SUB (X←X−Y)	SUB (X←Y−Z)
SUB	SUB M	MUL (X←X×Y)	MUL (X←Y×Y)
MUL	MUL M	DIV (X←X/Y)	DIV (X←Y/Z)
DIV	DIV M		

(8)多数指令集都有一条空操作(NOOP)指令。除了递增程序计数器之外,它对 CPU 状态没有任何影响。请列举这条指令的用途。

(9)假设一个栈被 CPU 用来管理过程调用和返回,能否以栈顶作为程序计数器从而取消原程序计数器?

(10)简述各种常用的寻址方式。

(11)使用变长指令的优缺点是什么?

(12)假设有一指令集,其指令长度是固定的 16 位,其中操作数字段为 6 位长。若有 K 条双操作数指令,L 条零操作数指令,那么该指令集能支持的单操作数指令的最大数目为多少?

(13)设计一种变长操作码,以允许如下指令均能编码成 36 位指令:指令有两个 15 位地址和一个 3 位寄存器号;指令有一个 15 位地址和一个 3 位寄存器号;指令没有地址或寄存器。

(14)什么是指令集结构的规整性?

(15)简述 CISC 指令集结构功能设计的主要目标。

(16)从当前的计算机技术观点来看,CISC 结构有什么缺点?

(17)简述 RISC 结构的设计原则。