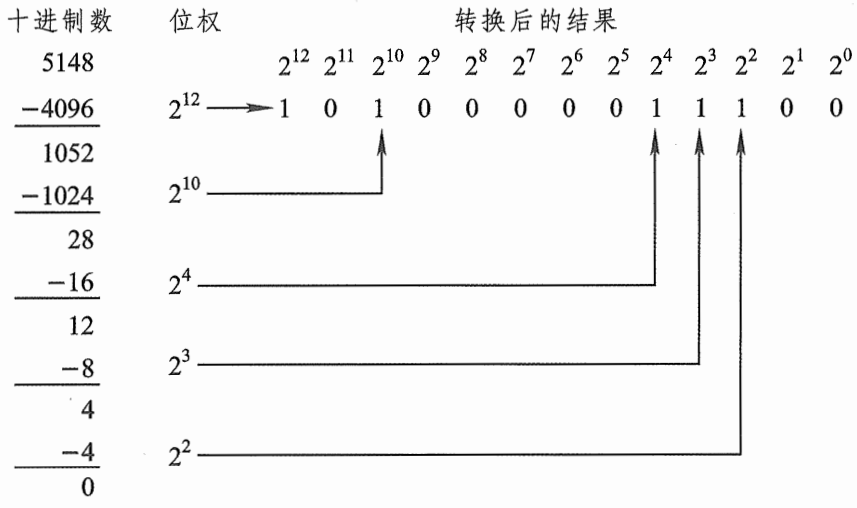
故小数部分$(0.6875)_{十}=(0.1011)_{二}$

所以　　$(123.6875)_{十}=(1111011.1011)_{二}$

(2) 减权定位法

当十进制数较大时,采用减权定位法可减少重复除法(或乘法)的次数。

例　将十进制数 5148 转换成二进制数,可按下列步骤进行。



即　　$(5148)_{十}=(1010000011100)_{二}$。

可见,只要从 5148 中减去所含的最大的 2 的方幂 4096,根据 $2^{12}=4096$,确定该权值对应的二进制数的位置(即数位),然后再从减得的 1052 中减去所含的最大的 2 的方幂 1024,又得该权值对应的数位,这样依次继续,直到差为 0。最后在有权值的对应数位上添 1,无权值的对应数位上添 0,便得转换结果。

3. 二进制数与八、十六进制数之间的转换

由于 $2^3=8$,$2^4=16$,故 3 位二进制数正好对应一位八进制数,4 位二进制数正好对应一位十六进制数,因此可按下述方法将二进制数转换成八、十六进制数。

例如:

1111000010.01101

↑

分组起点

高位补 0,凑足 3 位　　低位补 0,凑足 3 位

↑　　↑

001 111 000 010 . 011 010

1　7　0　2 . 3　2

故对应的八进制数为$(1702.32)_{八}=(1111000010.01101)_{二}$。

高位补 0,凑足 4 位　　低位补 0,凑足 4 位

0011	1100	0010	.	0110	1000
3	C	2	.	6	8

故对应的十六进制数为$(3C2.68)_{十六}=(1111000010.01101)_{二}$。

反之,将一位八进制数用 3 位二进制数表示,或一位十六进制数用 4 位二进制数表示,便可将八进制或十六进制数转换成二进制数。

例如:

$(247.63)_{八}=(10100111.110011)_{二}$

$(F5B.48)_{十六}=(111101011011.01001)_{二}$

附录 6B　阵列乘法器和阵列除法器

图 6.28 是一个完成 $X(X=X_1X_2X_3X_4)\times Y(Y=Y_1Y_2Y_3Y_4)$ 绝对值相乘的阵列乘法器原理图。图中方框的排列形式与笔算乘法的位积排列相似。阵列的每一行由乘数 Y 的每一位数位控制,而各行错开形成的每一斜列则由被乘数 X 的每一位数位控制。图中方框内的电路由一个与门和一个一位全加器组成。由于采用阵列结构,加法器数量很多,不靠“重复加和移位”的步骤运算,因此可大大提高乘法速度。该方案中虽然加法器数量多,但内部结构规则,标准化程度高,适

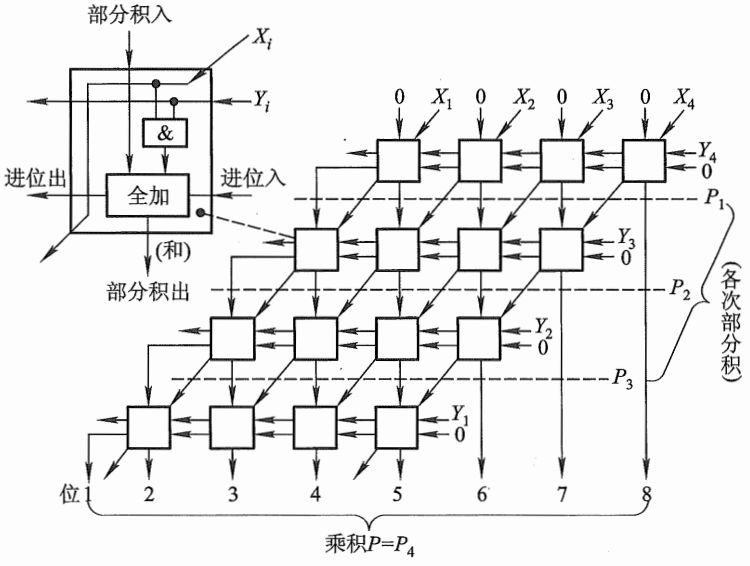
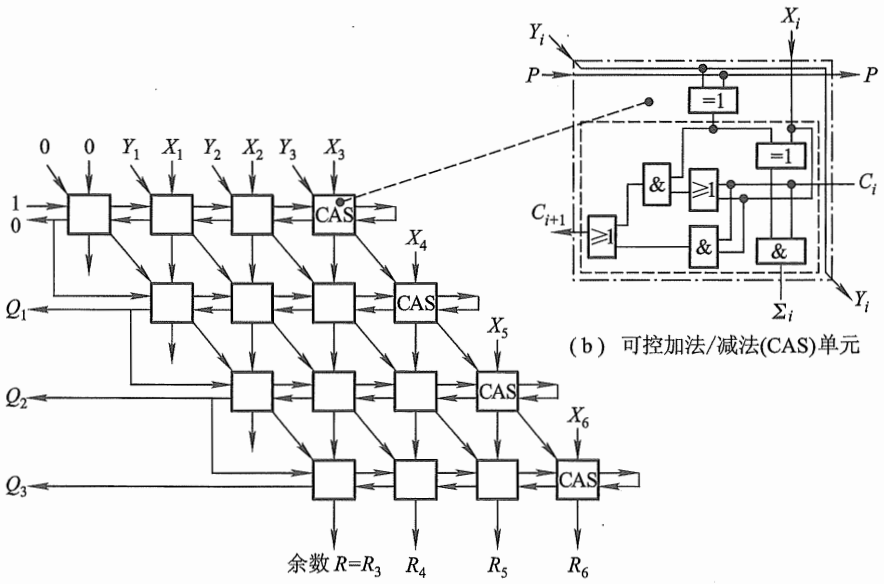


图 6.28　绝对值相乘的阵列乘法器

于用超大规模集成电路实现。

图 6.29 是一个完成 $X(X=X_1X_2X_3X_4X_5X_6)\div Y(Y=Y_1Y_2Y_3)$ 绝对值相除的不恢复余数除法器原理图。图中的每一个方框为一个可控加法/减法(CAS)单元，当输入控制 $P=0$ 时，CAS 做加法运算；当 $P=1$ 时，CAS 做减法运算。被除数 $X_1\sim X_6$ 由顶部一行和最右边对角线(斜列)上各 CAS 的垂直输入端提供；除数 $Y_1\sim Y_3$ 则沿对角线方向进入阵列，其作用是使余数固定(不左移)而除数右移，类似笔算除法；商 $Q_1Q_2Q_3$ 由阵列每一行最左边的 CAS 的进位输出 C_{i+1} 产生；余数 $R_3\sim R_6$ 在阵列的最下行产生。由于绝对值除和 6.3.4 节所述的原码除中数值部分的运算完全相同，故运算过程中需作 $X+Y$ 和 $X-Y$ 操作，而减法均用 $[|X|]_{补}+[-|Y|]_{补}$ 实现，因此阵列除法器中必有一些 CAS 单元用于对应符号位的运算，例如图 6.29 中每行最左边的小方框(共 4 个 CAS)。



(a) 4 位不恢复余数阵列除法器

(b) 可控加法/减法(CAS)单元

被除数 $X=0.X_1X_2X_3X_4X_5X_6$

除　数 $Y=0.Y_1Y_2Y_3$

商 $Q=0.Q_1Q_2Q_3$

余　数 $R=0.00R_3R_4R_5R_6$

图 6.29　绝对值相除的阵列除法器

以绝对值整数除法为例，第一步检查是否溢出，由第一行完成 $X_1X_2X_3-Y_1Y_2Y_3$ 操作，故控制电位 $P=1$。减法用 $[X]_{补}+[-|Y|]_{补}$ 实现，正好用 $P=1$ 作为第一行末位 CAS 的进位输入。由于 $X<Y$，所以相减后符号位的进位输出为 0，即商为 0，表示未溢出，除法可继续进行。此商接到第二行的 P 端，决定第二行做加法。同理每个当前商反馈到下一行，决定下一行是做加法还是减法，满足“上商 1 做减法，上商 0 做加法”的运算规则。

例　设 $x=101001$，$y=111$，用阵列除法器计算 $x\div y$。

解：$[x^*]_{补}=0,101001$；$[y^*]_{补}=0,111$；$[-y^*]_{补}=1,001$

被除数 x		0,101001	商		控制端
减除数 y	+	1,001			$P=1$
余数为负		1,110001	$Q_0=0$	未溢出	$P=0$
余数左移		1,10001			
加除数	+	0,111			
余数为正		0,01101	$Q_1=1$		$P=1$
余数左移		0,1101			
减除数	+	1,001			
余数为负		1,1111	$Q_2=0$		$P=0$
余数左移		1,111			
加除数	+	0,111			
		0,110	$Q_3=1$		

故商为 $Q_1Q_2Q_3=101$；余数为 $R_4R_5R_6=110$。

附录 6C　74181 逻辑电路

图 6.30 所示是 74181 ALU 的负逻辑电路，其逻辑关系如表 6.32 所示。

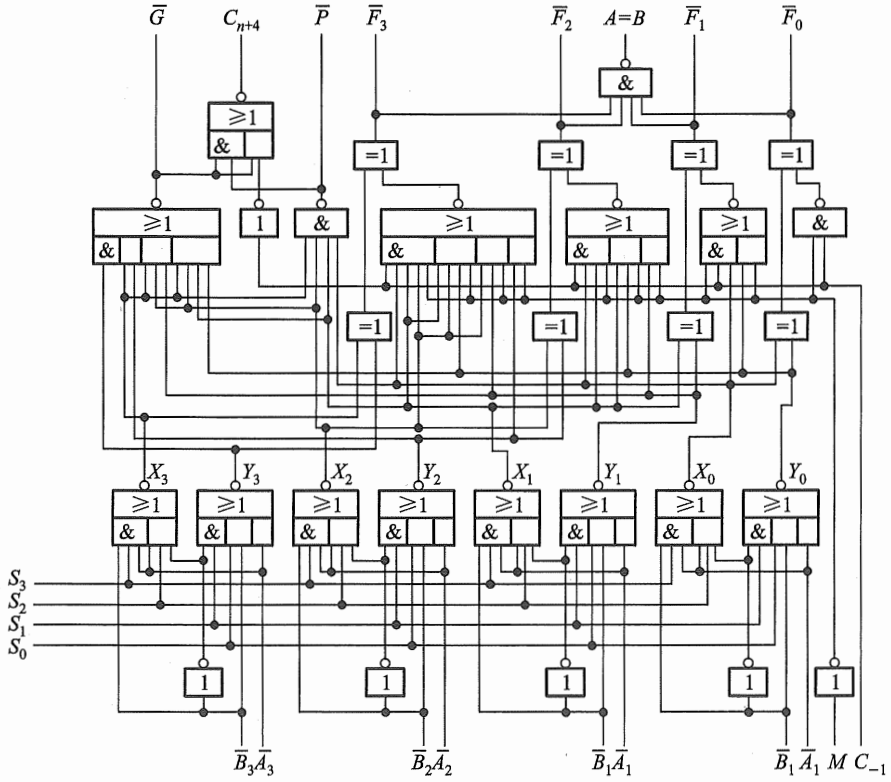


图 6.30　负逻辑操作数表示的 74181 ALU 电路

第7章　指令系统

本章主要介绍机器指令系统的分类、常见的寻址方式、指令格式以及设计指令系统时应考虑的各种因素。此外对 RISC 技术也进行简要的介绍，希望读者进一步体会指令系统与机器的主要功能以及与硬件结构之间存在的密切关系。

7.1　机器指令

由第 1 章可知，计算机能解题是由于机器本身存在一种语言，它既能理解人的意图，又能被机器自身识别。机器语言是由一条条语句构成的，每一条语句又能准确表达某种语义。例如，它可以命令机器做某种操作，指出参与操作的数或其他信息在什么地方等。计算机就是连续执行每一条机器语句而实现全自动工作的。人们习惯把每一条机器语言的语句称为机器指令，而又将全部机器指令的集合称为机器的指令系统。因此机器的指令系统集中反映了机器的功能。

计算机设计者主要研究如何确定机器的指令系统，如何用硬件电路、芯片、设备来实现机器指令系统的功能。计算机的使用者则是依据机器提供的指令系统，使用汇编语言来编制各种程序。计算机使用者根据机器指令系统所描述的机器功能，能很清楚地了解计算机内部寄存器-存储器的结构，以及计算机能直接支持的各种数据类型。

7.1.1　指令的一般格式

指令是由操作码和地址码两部分组成的，其基本格式如图 7.1 所示。

1. 操作码

操作码用来指明该指令所要完成的操作，如加法、减法、传送、移位、转移等。通常，其位数反映了机器的操作种类，也即机器允许的指令条数，如操作码占 7 位，则该机器最多包含 $2^7=128$ 条指令。

图 7.1　指令的一般格式

操作码的长度可以是固定的，也可以是变化的。前者将操作码集中放在指令字的一个字段内，如图 7.1 所示。这种格式便于硬件设计，指令译码时间短，广泛用于字长较长的、大中型计算机和超级小型计算机以及 RISC(Reduced Instruction Set Computer)中。例如，IBM 370 和 VAX-11 系列机，操作码长度均为 8 位。

对于操作码长度不固定的指令，其操作码分散在指令字的不同字段中。这种格式可有效地压缩操作码的平均长度，在字长较短的微型计算机中被广泛采用。例如 PDP-11、Intel 8086/80386 等，操作码的长度是可变的。

操作码长度不固定会增加指令译码和分析的难度，使控制器的设计复杂。通常采用扩展操作码技术，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，从而在满足需要的前提下，有效地缩短指令字长。图 7.2 是一种扩展操作码的安排示意图。

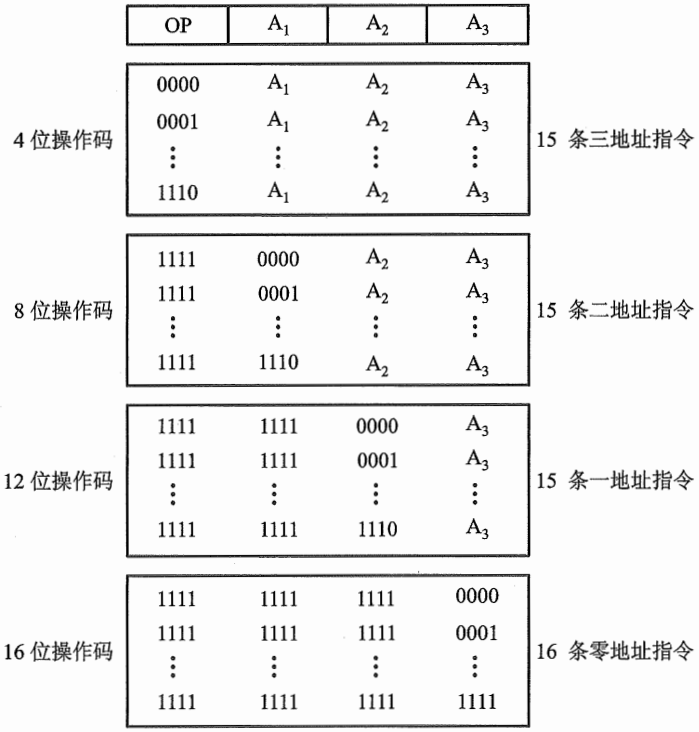


图 7.2　一种扩展操作码的安排示意图

图 7.2 中指令字长为 16 位，其中 4 位为基本操作码字段 OP，另有 3 个 4 位长的地址字段为 A_1、A_2、A_3。4 位基本操作码若全部用于三地址指令，则有 16 条。若采用扩展操作码技术，如图 7.2 所示，当操作码取 4 位时，三地址指令最多为 15 条；操作码取 8 位时，二地址指令最多为 15 条；操作码取 12 位时，一地址指令最多为 15 条；操作码取 16 位时，零地址指令为 16 条。共 61 条。可见操作码的位数随地址数的减少而增加。

除了这种安排以外，还有其他多种扩展方法，例如，形成 15 条三地址指令、12 条二地址指令、31 条一地址指令和 16 条零地址指令，共 74 条指令，读者可自行安排。

例 7.1　假设指令字长为 16 位，操作数的地址码为 6 位，指令有零地址、一地址、二地址三种

格式。

（1）设操作码固定，若零地址指令有 P 种，一地址指令有 Q 种，则二地址指令最多有几种？

（2）采用扩展操作码技术，若二地址指令有 X 种，零地址指令有 Y 种，则一地址指令最多有几种？

解：（1）根据操作数地址码为6位，则二地址指令中操作码的位数为 16-6-6=4。这4位操作码可有 $2^4=16$ 种操作。由于操作码固定，则除去了零地址指令 P 种，一地址指令 Q 种，剩下二地址指令最多有 $16-P-Q$ 种。

（2）采用扩展操作码技术，操作码位数可变，则二地址、一地址和零地址的操作码长度分别为4位、10位和16位。可见二地址指令操作码每减少一种，就可多构成 2^6 种一地址指令操作码；一地址指令操作码每减少一种，就可多构成 2^6 种零地址指令操作码。

因二地址指令有 X 种，则一地址指令最多有 $(2^4-X)\times 2^6$ 种。设一地址指令有 M 种，则零地址指令最多有 $[(2^4-X)\times 2^6-M]\times 2^6$ 种。

根据题中给出零地址指令有 Y 种，即

$$Y=[(2^4-X)\times 2^6-M]\times 2^6$$

则一地址指令

$$M=(2^4-X)\times 2^6-Y\times 2^{-6}$$

在设计操作码不固定的指令系统时，应尽量考虑安排指令使用频度（即指令在程序中出现的概率）高的指令占用短的操作码，对使用频度低的指令可占用较长的操作码，这样可以缩短经常使用的指令的译码时间。当然，考虑操作码长度时也应考虑地址码的要求。

2. 地址码

地址码用来指出该指令的源操作数的地址（一个或两个）、结果的地址以及下一条指令的地址。这里的“地址”可以是主存的地址，也可以是寄存器的地址，甚至可以是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

这种指令的地址字段有4个，其格式如下：

OP	A_1	A_2	A_3	A_4

其中，OP为操作码；A_1 为第一操作数地址；A_2 为第二操作数地址；A_3 为结果地址；A_4 为下一条指令的地址。

该指令完成 $(A_1)\,OP\,(A_2)\rightarrow A_3$ 的操作。这种指令直观易懂，后续指令地址可以任意填写，可直接寻址的地址范围与地址字段的位数有关。如果指令字长为32位，操作码占8位，4个地址字段各占6位，则指令操作数的直接寻址范围为 $2^6=64$。如果地址字段均指示主存的地址，则完成一条四地址指令，共需访问4次存储器（取指令一次，取两个操作数两次，存放结果一次）。

因为程序中大多数指令是按顺序执行的，而程序计数器PC既能存放当前欲执行指令的地址，又有计数功能，因此它能自动形成下一条指令的地址。这样，指令字中的第四地址字段 A_4 便

可省去，即得三地址指令格式。

(2) 三地址指令

三地址指令中只有 3 个地址，其格式如下：

OP	A_1	A_2	A_3

它可完成 $(A_1)\,OP\,(A_2)\rightarrow A_3$ 的操作，后续指令的地址隐含在程序计数器 PC 之中。如果指令字长不变，设 OP 仍为 8 位，则 3 个地址字段各占 8 位，故三地址指令操作数的直接寻址范围可达 $2^8=256$。同理，若地址字段均为主存地址，则完成一条三地址指令也需访问 4 次存储器。

机器在运行过程中，没有必要将每次运算结果都存入主存，中间结果可以暂时存放在 CPU 的寄存器(如 ACC)中，这样又可省去一个地址字段 A_3，从而得出二地址指令。

(3) 二地址指令

二地址指令中只含两个地址字段，其格式如下：

OP	A_1	A_2

它可完成 $(A_1)\,OP\,(A_2)\rightarrow A_1$ 的操作，即 A_1 字段既代表源操作数的地址，又代表存放本次运算结果的地址。有的机器也可以表示 $(A_1)\,OP\,(A_2)\rightarrow A_2$ 的操作，此时 A_2 除了代表源操作数的地址外，还代表中间结果的存放地址。这两种情况完成一条指令仍需访问 4 次存储器。如果使其完成 $(A_1)\,OP\,(A_2)\rightarrow ACC$，此时，它完成一条指令只需 3 次访存，它的含义是中间结果暂存于累加器 ACC 中。在不改变指令字长和操作码的位数前提下，二地址指令操作数的直接寻址范围为 $2^{12}=4$ K。

如果将一个操作数的地址隐含在运算器的 ACC 中，则指令字中只需给出一个地址码，构成一地址指令。

(4) 一地址指令

一地址指令的地址码字段只有一个，其格式如下：

OP	A_1

它可完成 $(ACC)\,OP\,(A_1)\rightarrow ACC$ 的操作，ACC 既存放参与运算的操作数，又存放运算的中间结果，这样，完成一条一地址指令只需两次访存。在指令字长仍为 32 位、操作码位数仍固定为 8 位时，一地址指令操作数的直接寻址范围达 2^{24}，即 16 M。

在指令系统中，还有一种指令可以不设地址字段，即所谓零地址指令。

(5) 零地址指令

零地址指令在指令字中无地址码，例如，空操作(NOP)、停机(HLT)这类指令只有操作码。而子程序返回(RET)、中断返回(IRET)这类指令没有地址码，其操作数的地址隐含在堆栈指针 SP 中(有关堆栈的概念详见 7.3.2 节)。

通过上述介绍可见，用一些硬件资源(如 PC、ACC)承担指令字中需指明的地址码，可在不改

变指令字长的前提下,扩大指令操作数的直接寻址范围。此外,用 PC、ACC 等硬件代替指令字中的某些地址字段,还可缩短指令字长,并可减少访存次数。因此,究竟采用什么样的地址格式,必须从机器性能出发综合考虑。

以上讨论的地址格式均以主存地址为例,实际上地址字段也可用来表示寄存器。当 CPU 中含有多个通用寄存器时,对每一个寄存器赋予一个编号,便可指明源操作数和结果存放在哪个寄存器中。地址字段表示寄存器时,也可有三地址、二地址、一地址之分。它们的共同点是,在指令的执行阶段都不必访问存储器,直接访问寄存器,使机器运行速度得到提高(因为寄存器类型的指令只需在取指阶段访问一次存储器)。

7.1.2 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相同的。

早期的计算机指令字长、机器字长和存储字长均相等,因此访问某个存储单元,便可取出一条完整的指令或一个完整的数据。这种机器的指令字长是固定的,控制方式比较简单。

随着计算机的发展,存储容量的增大,要求处理的数据类型增多,计算机的指令字长也发生了很大的变化。一台机器的指令系统可以采用位数不相同的指令,即指令字长是可变的,如单字长指令、多字长指令。控制这类指令的电路比较复杂,而且多字长指令要多次访问存储器才能取出一条完整的指令,因此使 CPU 速度下降。为了提高指令的运行速度和节省存储空间,通常尽可能把常用的指令(如数据传送指令、算逻运算指令等)设计成单字长或短字长格式的指令。

例如,PDP-8 指令字长固定取 12 位;NOVA 指令字长固定取 16 位;IBM 370 指令字长可变,可以是 16 位(半个字)、32 位(一个字)、48 位(一字半);Intel 8086 的指令字长可以为 8、16、24、32、40 和 48 位六种。通常指令字长取 8 的整数倍。

7.2 操作数类型和操作类型

7.2.1 操作数类型

机器中常见的操作数类型有地址、数字、字符、逻辑数据等。

(1) 地址

地址实际上也可看作是一种数据,在许多情况下要计算操作数的地址。这时,地址可被认为是一个无符号的整数,有关地址的计算问题将在 7.3 节讨论。

(2) 数字

计算机中常见的数字有定点数、浮点数和十进制数。前两种数字在第6章中已进行了介绍，十进制数已在第5章附录中说明，读者可自行复习。

(3) 字符

在应用计算机时，文本或者字符串也是一种常见的数据类型。由于计算机在处理信息过程中不能以简单的字符形式存储和传送，因此普遍采用ASCII码(见表5.2)，它是很重要的一种字符编码。当然还有其他一些字符编码，如8位EBCDIC码(Extended Binary Coded Decimal Interchange Code)，又称扩展BCD交换码，在此不做详述。

(4) 逻辑数据

计算机除了做算术运算外，有时还需做逻辑运算，此时 n 个0和1的组合不是被看作算术数字，而是被看作逻辑数。例如，在ASCII码中的0110101，它表示十进制数5，若要将它转换为NBCD短十进制码，只需通过它与逻辑数0001111完成逻辑与运算，抽取低4位，即可获得0101。此外，有时希望存储一个布尔类型的数据，它们的每一位都代表着真(1)和假(0)，这时 n 个0和1组合的数就都被看作逻辑数。

例如，奔腾处理器的数据类型有逻辑数、有符号数(补码)、无符号数、压缩和未压缩的BCD码、地址指针、位串、字符串以及浮点数(符合IEEE 754标准)等。

7.2.2 数据在存储器中的存放方式

通常计算机中的数据存放在存储器或寄存器中，而寄存器的位数便可反映机器字长。一般机器字长可取字节的1、2、4、8倍，这样便于字符处理。在大、中型机器中字长为32位和64位，在微型计算机中字长从4位、8位逐渐发展到目前的16位、32位和64位。

由于不同的机器数据字长不同，每台机器处理的数据字长也不统一，例如奔腾处理器可处理8(字节)、16(字)、32(双字)、64(四字)；PowerPC可处理8(字节)、16(半字)、32(字)、64(双字)。因此，为了便于硬件实现，通常要求多字节的数据在存储器的存放方式能满足“边界对准”的要求，如图7.3所示。

图7.3中所示的存储器存储字长为32位，可按字节、半字、字、双字访问。在对准边界的32位字长的计算机中(如图7.3(a)所示)，半字地址是2的整数倍，字地址是4的整数倍，双字地址是8的整数倍。当所存数据不能满足此要求时，可填充一个至多个空白字节。而字节的次序有两种，如图7.4所示，其中7.4(a)表示低字节为低地址，图7.4(b)表示高字节为低地址。

在数据不对准边界的计算机中，数据(例如一个字)可能在两个存储单元中，此时需要访问两次存储器，并对高低字节的位置进行调整后才能取得一个字，图7.3(b)的阴影部分即属于这种情况。

图 7.3　存储器中数据的存放

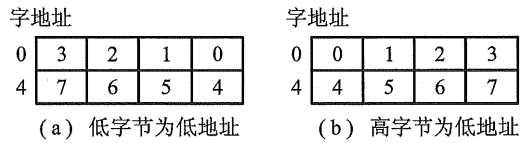


图 7.4　两种字节次序

7.2.3　操作类型

不同的机器,操作类型也是不同的,但几乎所有的机器都有以下几类通用的操作。

1. 数据传送

数据传送包括寄存器与寄存器、寄存器与存储单元、存储单元与存储单元之间的传送。如从源到目的之间的传送、对存储器读(LOAD)和写(STORE)、交换源和目的的内容、置 1、清零、进栈、出栈等。

2. 算术逻辑操作

这类操作可实现算术运算(加、减、乘、除、增 1、减 1、取负数即求补)和逻辑运算(与、或、非、

异或)。对于低档机而言,一般算术运算只支持最基本的二进制加减、比较、求补等,高档机还能支持浮点运算和十进制运算。

有些机器还具有位操作功能,如位测试(测试指定位的值)、位清除(清除指定位)、位求反(对指定位求反)等。

3. 移位

移位可分为算术移位、逻辑移位和循环移位三种。算术移位和逻辑移位分别可实现对有符号数和无符号数乘以 2^n(左移)或整除以 2^n(右移)的运算,并且移位操作所需时间远比乘除操作执行时间短,因此,移位操作经常被用来代替简单的乘法和除法运算。

4. 转移

在多数情况下,计算机是按顺序执行程序的每条指令的,但有时需要改变这种顺序,此刻可采用转移类指令来完成。转移指令按其转移特征又可分为无条件转移、条件转移、跳转、过程调用与返回、陷阱(Trap)等几种。

(1) 无条件转移

无条件转移不受任何条件约束,可直接把程序转移到下一条需执行指令的地址。例如"JMP X",其功能是将指令地址无条件转至 X。

(2) 条件转移

条件转移是根据当前指令的执行结果来决定是否需要转移。若条件满足,则转移;若条件不满足,则继续按顺序执行。一般机器都能提供一些条件码,这些条件码是某些操作的结果。例如:零标志位(Z),结果为 0,Z=1;负标志位(N),结果为负,N=1;溢出标志位(V),结果有溢出,V=1;进位标志位(C),最高位有进位,C=1;奇偶标志位(P),结果呈偶数,P=1 等。

例如,指令"BRO X"表示若结果(有符号数)溢出(V=1),则指令跳转至 X。例如,指令"BRC Y"表示若最高位有进位(C=1),则指令跳转至 Y。

还有一种条件转移指令,SKP(Skip),它暗示其下一条指令将被跳过,从而隐含了转移地址是 SKP 后的第二条指令。例如:

```
200
 ⋮
205 SKP DZ
206
207
```

这里"SKP DZ"表示若设备的完成触发器 D 为零,则执行完 205 条指令后,立即跳至第 207 条指令,再顺序执行。

(3) 调用与返回

在编写程序时,有些具有特定功能的程序段会被反复使用。为避免重复编写,可将这些程序段设定为独立子程序,当需要执行某子程序时,只需用子程序调用指令即可。此外,计算机系统还提供了通用子程序,如申请资源、读写文件、控制外设等。需要时均可由用户直接调用,不必重

新编写。

通常调用指令包括过程调用、系统调用和子程序调用。它可实现从一个程序转移到另一个程序的操作。

调用指令(CALL)一般与返回指令(RETURN)配合使用。CALL 用于从当前的程序位置转至子程序的入口;RETURN 用于子程序执行完后重新返回到原程序的断点。图 7.5 示意了调用(CALL)和返回(RETURN)指令在程序执行中的流程。

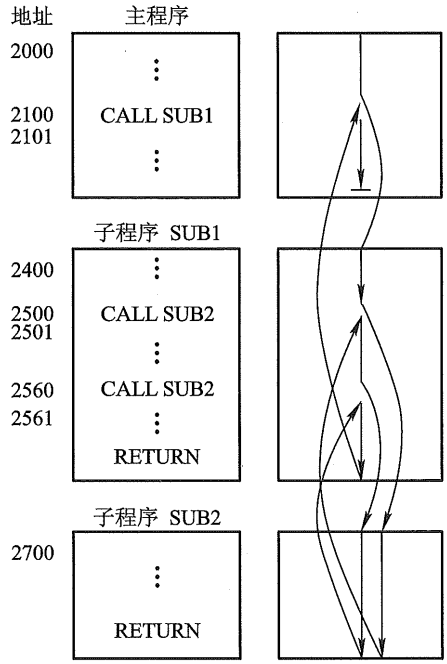


图 7.5　调用和返回指令示意图

图 7.5(a)示意了主程序和子程序在主存所占空间。主程序从 2000 地址单元开始,并在 2100 处有一个调用指令,当执行到 2100 处指令时,CPU 停止下一条顺序号为 2101 的指令,而转至 2400 执行 SUB1 子程序。在 SUB1 中又有两次(2500 和 2560 处)调用子程序 SUB2。每一次都将 SUB1 挂起,而执行 SUB2。子程序末尾的 RETURN 指令可使 CPU 返回调用点。

图 7.5(b)示意了主程序→SUB1→SUB2→SUB1→SUB2→SUB1→主程序的执行流程。

需要注意以下几点。

- 子程序可在多处被调用。
- 子程序调用可出现在子程序中,即允许子程序嵌套。
- 每个 CALL 指令都对应一条 RETURN 指令。

由于可以在许多处调用子程序,因此,CPU 必须记住返回地址,使子程序能准确返回。返回地址可存放在以下 3 处。

- 寄存器内。机器内设有专用寄存器,专门用于存放返回地址。
- 子程序的入口地址内。
- 栈顶内。现代计算机都设有堆栈,执行 RETURN 指令后,便可自动从栈顶内取出应返回的地址。

(4) 陷阱(Trap)与陷阱指令

陷阱其实是一种意外事故的中断。例如,机器在运行中,可能会出现电源电压不稳定、存储器校验出差错、输入输出设备出现了故障、用户使用未被定义的指令、除数出现为 0、运算结果溢出以及特权指令等种种意外事件,致使计算机不能正常工作。此刻必须及时采取措施,否则将影响整个系统的正常运行。因此,一旦出现意外故障,计算机就发出陷阱信号,暂停当前程序的执行,转入故障处理程序进行相应的故障处理。

计算机的陷阱指令一般不提供给用户直接使用,而作为隐指令(即指令系统中不提供的指令),在出现意外故障时,由 CPU 自动产生并执行。也有的机器设置供用户使用的陷阱指令或“访管”指令,利用它完成系统调用和程序请求。例如,IBM PC(Intel 8086)的软中断 INT TYPE(TYPE 是 8 位常数,表示中断类型),其实就是直接提供给用户使用的陷阱指令,用来完成系统调用。

5. 输入输出

对于 I/O 单独编址的计算机而言,通常设有输入输出指令,它完成从外设中的寄存器读入一个数据到 CPU 的寄存器内,或将数据从 CPU 的寄存器输出至某外设的寄存器中。

6. 其他

其他包括等待指令、停机指令、空操作指令、开中断指令、关中断指令、置条件码指令等。

为了适应计算机的信息管理、数据处理及办公自动化等领域的应用,有的计算机还设有非数值处理指令。如字符串传送、字符串比较、字符串查询及字符串转换等。

在多用户、多任务的计算机系统中,还设有特权指令,这类指令只能用于操作系统或其他系统软件,用户是不能使用的。

在有些大型或巨型机中,还设有向量指令,可对整个向量或矩阵进行求和、求积运算。在多处理器系统中还配有专门的多处理机指令。

7.3 寻址方式

寻址方式是指确定本条指令的数据地址以及下一条将要执行的指令地址的方法,它与硬件结构紧密相关,而且直接影响指令格式和指令功能。

寻址方式分为指令寻址和数据寻址两大类。