第十二章重点讨论了指令集做什么，特别是考察了机器指令可指定的操作和操作数类型。本章讨论如何指定指令的操作和操作数。这里要解决两个问题，首先是如何指定操作数地址，其次是指令的位如何组织，以定义操作数地址和操作。

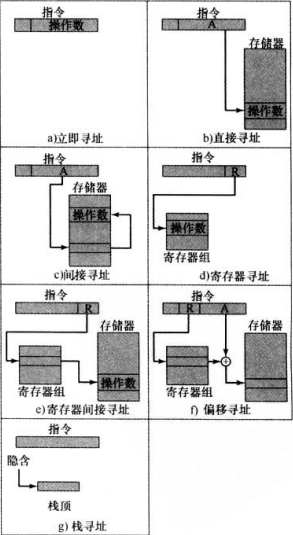


图13-1 寻址方式

图13-1 寻址方式

13.1 寻址方式

正如以前提到过的，指令格式中的地址字段通常是相对较小的。我们希望，有能力大范围地访问主存或虚拟存储器。为实现此目标，指令采用了各类寻址技术。他们都涉及地址范围和寻址灵活性之间，以及储存器引用数和地址计算复杂性之间的权衡考虑。本节将考察最常用的寻址技术：立即寻址、直接寻址、间接寻址、寄存器寻址、寄存器间接寻址、偏移寻址、栈寻址。

图13-1展示了这些方式。在本节中，我们将使用如下表示

法：

A=指令中地址字段的内容

R=指向寄存器的指令地址字段内容

EA=被访问位置的实际（有效）地址

（x）=被访问位置X或寄存器X的内容

表13-1列出了每种寻址方式所进行的地址计算

在讨论之前需要说明两点。首先，实际上所有计算机结构都需要提供不止一种寻址方式。这也就提出一个问题，处理器如何确认什么样的寻址方式正被一条指令所使用。这有几种方法：通常是不同操作码使用不同的寻址方式。另外，指令格式中的一位或几位能用做*方式字段*（mode field），方式字段的值确定使用哪种寻址方式。

第二点说明是关于有效地址（EA）的解释。在没有虚拟存储器的系统中，*有效地址*（effective address）将是主存地址或是寄存器。在虚拟存储器系统中。有效地址是虚拟地址或是寄存器。把虚拟地址映射到物理地址实际上是内存管理单元（MMU）的功能，并且是程序员不可见的。

表13-1 基本寻址方式



13.1.1 立即寻址

寻址的最简单形式是*立即寻址*（immediate addressing）。在这种方式下，操作数实际出现在指令中：

操作数 = A

这种方式能用于定义和使用常数或者设置变量的初始值。一般地，数字以2的补码形式储存，最左位是符号。当操作数装入数据寄存器时，符号位向左扩展来填充数据字的字长。在某些情况下，立即二进制值被当作无符号非负整数。

立即寻址的优点是，除了取指令之外，获得操作数不要求另外的储存访问，于是节省了一个存储器或高速缓存（cache）周期。缺点是数的大小受限于地址字段的长度，而在大多数指令集中此字段长度与字长度相比是比较短的。

13.1.2 直接寻址

直接寻址（direct addressing）也是一种很简单的寻址形式，这种方式下地址字段含有操作数的有效地址：

EA=A

此技术在早期计算机中是很普遍的，但在当代计算机体系结构中就不多见了。它只要求一次存储器的访问，而且不需要生成地址的专门计算。正如前面所述，明显的不足是只能提供有限的地址空间。

13.1.3 间接寻址

直接寻址的问题是地址字段的长度通常小于字长度，这样寻址范围就很有限，一个解决方法是，让地址字段指示一个存储器字地址，而此地址处保存有操作数的全长度地址。这被称为间接寻址（indirect addressing）:

EA=(A)

正如前面所说明的，括号解释成“其内容”（content of）。这种方法的明显优点是，对于N位字长来说能有2N个地址可用。缺点是为取一操作数，指令执行需要两次访问存储器：第一次为得到地址，第二次才是得到它的值。

虽然能被寻址的字的数目现在等于2N，但一次能被引用的不同有效地址数目限制到2K，这里K是地址字段的位长度。一般而言，这并不是一个严重的限制，而且有可能是有益的。在虚拟存储器环境中，所有的有效地址位置都能限定放到任何进程的第0页内。因为指令的地址字段通常比较小，自然会形成数目不多且数值不大的直接地址，这些地址对应的储存位置可以放在第0页中（唯一的限制是页的大小必须大于或等于2K）。当一个进程激活时，将会有对第0页的重复访问，这将使它保留在物理存储器中。于是，一次间接存储器访问最多只涉及一次缺页而不是两次。

一种很少使用的间接寻址的变体是多级或级联的间接寻址：

EA=（…（A）…）

这种情况下，全字地址中有一个间接标志位（I）。若此位是0，则全字地址中的其余位是有效地址EA。若I位是1，则要求另一极的间接。这种方法没什么特别的好处，缺点是为取一操作数要求三次甚至更多次的存储器访问。

13.1.4 寄存器寻址

寄存器寻址（register addressing）类似于直接寻址。唯一的不同是地址字段指的是寄存器而不是一个主存地址：

EA= R

举例说明一下，假设指令中寄存器地址字段的值是5，那么寄存器R5就是所指定的地址，操作数的值就是R5的内容。一般的，引用寄存器的地址字段有3到5位，因此能访问总计8~32个通用寄存器。

寄存器寻址的优点，一是指令中仅需要一个较小的地址字段，二是不需要存储器访问。正如第四章讨论过的，对CPU内部寄存器存取的时间是远小于主存存取时间的。寄存器寻址的缺点是地址空间十分有限。

若指令集中大量地使用了寄存器寻址，这意味着CPU寄存器被大量使用。因为寄存器数量及其有限（与主存位置相比），所以只有他们能得到有效使用的应用才有意义。若是每个操作数都由主存来装入寄存器，操作一次后又送回主存，则暂存这些内容实际上是一种浪费。然而，若是留在寄存器中的操作数能为多个操作数所使用，则实现了有效的节省，例如计算的中间结果。具体的，假设2的补码数的乘法是以软件实现的，那么，图10-12流程图中标记为A的位置是要多次访问的，因此应以寄存器而不是主存位置来实现。

哪些值应保留在寄存器中，哪些值应存于储存器，这个判决应由程序员或编译器完成。大多数当代CPU都使用多个通用寄存器，如果有效地使用它们就成为汇编语言编程人员（例如，编译器的编写者）的责任。

13.1.5 寄存器间接寻址

正如寄存器寻址类似于直接寻址一样，寄存器间接寻址（register indirect addressing）也类似于间接寻址。两种情况的唯一不同是，地址字段指的是储存位置还是寄存器，于是，对于寄存器间接寻址：

EA=（R）

寄存器间接寻址的优点和不足基本上与间接寻址类似，二者的地址空间限制（有效的地址范围）都通过将地址字段指向一个保存有全长地址的位置而被克服了，另外，寄存器间接寻址比间接寻址少一次存储器的访问。

13.1.6 偏移寻址

一种强有力的寻址方式是直接寻址和寄存器间接寻址能力的结合。根据上下文的不同，它有几种名称，但基本的机制是相同的。我们将它称为便宜寻址（displacement addressing）。

EA=A+（R）

偏移选址要求指令由两个地址字段，至少其中一个是显式的，保存在一个地址字段中的值（值=A）直接被使用。另一个地址字段，或者一个基于操作码的隐含引用，指向一个寄存器。此寄存器的内容加上A产生有效地址。

我们将介绍最通用的三种偏移寻址：

× 相对寻址

× 机制寄存器寻址

× 变址

1、相对寻址

对于相对寻址，隐含引用的寄存器是程序计数器（PC），因此也叫PC相对寻址。即当前PC的值（此指令后续的下一条指令的地址），加上地址字段的值（A），产生有效地址。一般地，地址字段的值这种操作下被看成是2的补码数的值，于是，有效地址是对当前指令地址的一个前后范围的偏移。

相对寻址利用了第四章和第八章讨论过的局部性概念。如果大多数存储器访问都相对于靠近正在执行的指令，则使用相对寻址可节省指令中的地址位数。

2、基址寄存器寻址

对于这种寻址方式，其解释如下：被引用的寄存器含有一个存储器地址，地址字段含有一个相对于那个地址的偏移量（通常是无符号整数表示）。寄存器引用可以使显式的，也可以是隐式的。

基址寄存器寻址也利用了存储器访问的局部性。用它来实现第八章讨论过的段是一种方便的方式。在某些方案中。采用一个单一的段基址寄存器，并且是隐含使用。而在其他情况中，程序员可选取一个寄存器来保留段的基地址，并且指令对他必须显式引用。在后一种情况下，若地址字段的位长度是K，并且可选取的寄存器有N个，则一条指定能访问N\*2K个字的域中的任一字。

3、变址

对于变址，典型的解释如下：指令地址字段引用一个主存地址，被引用的寄存器含有对于那个地址的一个正的偏移量。注意，这种用法正好和基址寄存器寻址方式的解释相反。当然，两者的区别并不只是一个用法解释的问题。因为变址中的地址字段将被看做一个存储器地址。所以该地址字段通常要比基址寄存器指令中的地址字段包含更多的位。还有，我们将会看到对于变址将会有某些改进，而这些改进不能用于基址寄存器的方式。不过无论如何，基址寄存器寻址和变址两者的EA计算方法是相同的，而且两种情况下的寄存器引用都是有时是显示，有时是隐式的（对于不同类型的CPU）。

变址的一个重要用途是为重复操作的完成提供一种高效机制。例如，在位置A处开除开始有一个数值列表，我们准备为表的每个元素加1，这需要取出表中的每个数值，对他加1，然后再存回。需要的有效地址序列是A，A+1，A+2，…，直至表的最后一个位置。使用变址，这很容易完成。将值A存入指令的地址字段。并选取一个寄存器，叫做变址寄存器（index register），初始化为0.每次操作之后，变址寄存器加1。

因为变址寄存器普遍用于这种重复任务，故一般都需要对它每次引用之后将它递增或递减。某些系统是自动完成这种递增递减的操作，并将其作为同一指令周期的一部分，这称为自动变址（autoindexing）。若某个寄存器专门用于变址，则自动变址能隐含地自动启动。若使用通用寄存器作为变址寄存器，则自动变址操作需要用指令中的某一位来通知。采用递增的自动变址可描述为如下：

EA=A +（R）

（R）←（R）+1

某些机器既提供了间接寻址又提供了变址，在同一指令中使用两种寻址方式是准许的。这有两种可能的方式：在间接寻址之前或之后进行变址寻址。

如果在间接寻址之后进行变址，这称为后变址（post-indexing）：

EA=（A）+（R）

首先，地址字段的内容用来访问一个存储器位置取得直接地址。然后，这个地址被寄存器值变址。对于存取若干具有固定格式数据块中的某一个，这种技术是很有用的。例如，在第八章曾讨论过，操作系统需要为每个进程访问进程控制块。不管当前正在操作哪个块，完成的操作都是相同的。于是，访问这些块的指令中的那些地址（值=A）可以都指向一个保存有一个可变指针的位置，此可变指针指向具体进程控制块的起点。变址寄存器中保存有块内偏移量。

前变址（pre-indexing）是变址完成在间接寻址之前：

EA=（A+（R））

像简单变址一样，完成一次地址计算，然而所计算出的地址含有的不是操作数而是操作数的地址。这种技术使用的一个例子是构建多跳转表（multiway branch table）。在程序的某一点上，可能要根据一些条件转移到几个位置中的某一个。可以用位置A作为起点建立一个地址表，通过变址到这个表中，来找到所要求的位置。

正常情况下，指令集将不会同时包括前变址和后变址。

13.1.7 栈寻址

最后考虑栈寻址（stack addressing）方式。正如附录1中所定义的，栈是一种位置的线性序列。它有时称为下推表（pushdown list）或后进先出队列（last-in-first-out queue）。栈是一个预留的位置块。数据项是被陆续加到栈顶，因此在任一给定时刻，栈对应的位置块是部分被填充的。与栈相关的是一个指针，它的值是栈顶地址。或者，当栈顶的两个元素已在CPU寄存器内，此时栈指针指向栈顶的第三个元素。栈指针保存在寄存器中，于是对储存器中的栈位置的访问实际上是一种寄存器间接寻址方式。

栈寻址方式是一种隐含寻址方式。机器指令不需要指明存储器引用，而是隐含地指示操作发生在栈顶。

13.2 x86和ARM寻址方式

13.2.1 x86寻址方式

我回顾图8-21，x86的地址转换机制产生一个地址，称为虚拟地址或有效地址，它是一个段内位移（offest）。段的起始地址和这个有效地址之和就构成一个线性地址（linear address）。如果采用了分页，这个线性地址必须通过一个页转换机制来生成一个物理地址。下面暂不管这最后一步，因为它对于指令集和程序员都是透明的。

x86配备了各种寻址方式，目的是使高级语言能够有效的执行。图13-2指出了所涉及的硬件逻辑。被访问的对象是段，它由段寄存器所确定。有6个段寄存器，具体的访问使用哪一个段寄存器，这取决于执行的上下文和指令。每个段寄存器保存一个指向段描述符表（segment descriptor table）的索引（见图8-20），段描述符表保存了各个段的起始地址。每个段寄存器（用户可见的）与一个段描述符寄存器（程序员不可见的）关联，段描述符寄存器记录了此段的访问权限，以及段的起始地址和段的界限（段的长度）。此外，还有两个寄存器（基址寄存器和变址寄存器），可用于构造地址。

表13-2列出了x86的寻址方式，让我们依序考察每一种方式。

对于立即方式（immediate mode），操作数被包含在指令中。操作数可以是字节、字或双字的数据。

对于寄存器操作数方式（register operand mode），操作数位于寄存器中。对于像数据传送、算术和逻辑指令这样的一般指令，操作数可以是一个32位通用寄存器（EAX、EBX、ECX、EDX、ESI、EDI、ESP和EBP）、一个16位通用寄存器（AX、BX、CX、DX、SI、DI、SP和 BP），或是一个8位通用寄存器（AH、BH、CH、DH、AL、BL、CL和DL）。还有访问段寄存器（CS、DS、ES、SS、FS和GS）的一些指令。

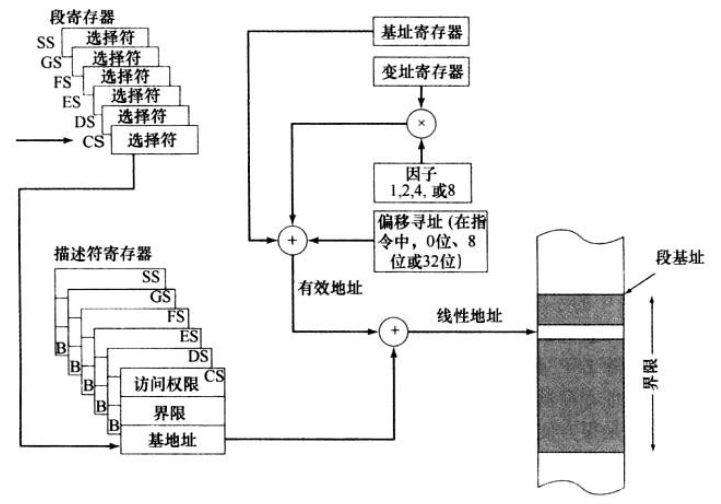




表13-2 x86寻址方式

其余的寻址方式引用的是储存器中的位置。通过指定包含由此位置的段和距离段起点的位置来说明存储器位置。某些情况下，段是显示指定的。在其他情况下，段通过一个简单的段默认指派规则来隐式指定。

在偏移量方式（displacement mode）中，指令中的8位、16位或32位偏移量就是操作数距段起点的位移。偏移量寻址方式只在少数几种机器中能找到，因为它会导致较长的指令。在x86机器中，偏移量值可达32位，这就使指令多达6字节长。偏移量寻址方式对于访问全局变量很有用。

剩下的寻址方式是指令的地址部分告诉处理器到何处去找地址，因此，他们是间接方式。基址方式指定一个8位、16位或32位的寄存器，其中包含有效地址。这等同于我们已说过的寄存器间接寻址。

在基址带偏移量方式（base with displacement mode）中，指令包括一个将被加到基址寄存器的偏移量，基址寄存器可是任意一个通用寄存器。这种方式的使用例子有：

× 编译器用来指向一个局部变量域的开始。例如，基址寄存器能用来指向栈帧的起点，而栈帧含有相应过程的局部变量。

× 当数组元素大小不是1、2、4或8字节，从而不能使用变址寄存器来索引时，可用这种方式来索 引数组。此时，偏移量指向数组起点，基址寄存器保存指定数组元素距数组起点的位移。

× 用于访问记录中的字段，基址寄存器指向记录的起点，而偏移量是到此字段的位移。

在比例变址带偏移量方式（scaled index with displacement mode）中，指令包括一个将加到变址寄存器的偏移量。除ESP通常用于栈处理之外，其他任何通用寄存器都可以作为变址寄存器。计算有效机地址时，变址寄存器的内容乘以1、2、4或8的比例因子，然后加上偏移量。对于索引一个数组，这种方式是很方便的。比例因子为2能用于一个16位整数数组，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的数组。

基址带变址和偏移量方式（base with index and displacement mode）是将基址寄存器内容、变址寄存器内容和偏移量三者求和，得到有效地址。同样，除ESP通常用于栈处理之外，其他任何通用寄存器都可以作为基址和变址寄存器。作为一个例子，这种寻址方式可以用于存取栈帧中的局部数组、除此之外，它也可以用于寻址二维数组，此时偏移量指向数组的起点，基址和变址寄存器分别处理二维数组中一个维的地址。

基址的比例变址带偏移量方式（based scaled index with displacement mode）是将变址寄存器内容乘以比例因子、基址寄存器内容和偏移量三者求和。若一个数组存于栈帧中，这种寻址方式是很有用的，此时数组元素可以是2、4或8字节长。这种方式亦能对数组元素是2、4或8字节长的二维数组提供有效的索引。

最后，相对寻址（relative addressing）能用于控制转移指令。一个偏移量加到指令下一条指令的程序计数器的值上。此时，偏移量被看作是一个有符号的字节、字和双字值，于是此值能增加或减少程序计数器中的地址。

13.2.1 ARM寻址方式

不像CISC机器，RISC机器一般都普遍采用简单和相对直截了当的一组寻址方式。不过ARM与这些传统有些不同，它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

1、装载/保存寻址

ARM中只有装载/保存（Load/Store）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3种不同方式（见图13-3）。

× 偏移（offest）：对于这种寻址方式，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。例如，图13-3a显示了采用这种寻址方式的汇编语言指令STRB r0,[r1,#12]（字体格式对吗？）。这条指 令保存一个字节到内存中，在这个例子中，基址在寄存器r1中，偏移量是一个立即值，十进制数12。得到的地址（基址加上偏移量）就是r0的最低有效字节所要保存数据的位置。

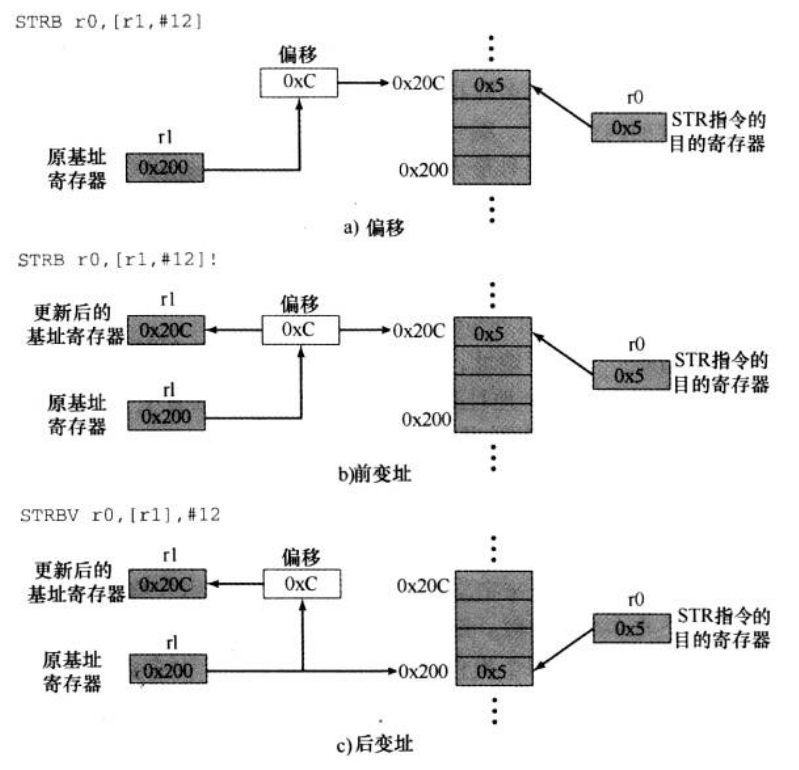


图13-3 ARM变址方式

× 前变址（preindex）：内存地址的计算与上面偏移寻址的方式一样。然后，计算得到的内存地址被写回到基址寄存器。也就是说，基址寄存器的值增加或减少了一个偏移量的值。图13-3b显示了采用这种寻址方式的汇编语言指令STRB r0,[r1, #12]!。其中“!”表示要进行的前变址。

× 后变址（postindex）：内存地址就是基址寄存器的值，然后，偏移量被加到基址寄存器值中，或从中减去，结果再保存回基址寄存器。图13-3c显示了采用这种寻址方式的汇编语言指令STRB r0, #12。

可以注意到,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：带比例的寄存器寻址。在偏移量寄存器中保存的值是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，算数右移，循环右移，以及扩展的循环右移（在循环移位中包括进位位）。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上文介绍的5种移位操作来进行按比例的放大和缩小。

3、分支指令

分支指令只有一种寻址方式：立即寻址。分支指令中带有一个24位的立即值。在地址计算时，这个立即值被左移2位，使得地址对齐到32位字的边界。这样，24位的立即值可以产生26位的地址偏移量，相对于程序计数器来说，（第十版不存在这句话）有效地址的范围为+-（正负符号啊）32MB。

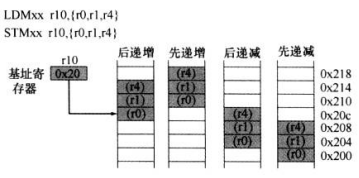


图13-4 ARM多装/多存寻址

图13-4 ARM多装/多存寻址

４、多装/多存寻址

多装（Load Multiple）指令从内存装载多个数据到多个（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。用作装载或保存的寄存器列表由指令中一个16位的字段给出，其中每一位对应的16个寄存器中的一个。多装和多存指令寻址方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（见图13-4）：后递增、先递增、后递减、先递减。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和之前。

这些指令对于数据块装载和保存、栈操作以及过程退出操作都是很有用的。

13.3 指令格式

指令格式通过它的各个构成部分来定义指令的位安排。一个指令格式必须包括一个操作码，以及隐式或显式的、零个或多个操作数。每个显式操作数使用13.1节描述的某种寻址方式来引用。指令格式必须显示或隐式地为每个操作数指定其寻址方式。大多数指令集使用不止一种指令格式。

指令格式设计是一种复杂的技术，已有的指令格式种类之繁多令人吃惊。本节考察关键的设计出发点，通过简要地查看某些设计来说明这些出发点。然后，再详细考察x86和ARM的做法。

13.3.1 指令长度

设计人员面对的最基本设计出发点，是指令格式的长度。这个决定与储存器尺寸、储存器组织、总线结构、CPU复杂程度和细CPU速度等相关影响。它决定了汇编语言编程人员所看到的机器指令的丰富性和灵活程度。

此时最明显的权衡考虑是在强有力的指令清单和必须节省空间之间进行。编程人员希望更多的操作码、更多的操作数、更多的寻址方式和更大的地址范围。更多的操作码和更多的操作数可使编程人员的日子更好过，他们能写出较短的程序来完成给定的任务。类似地，更多的寻址方式可给操作人员在实现某些像表处理和跳转这样的功能时有更大的灵活度。还有，随着主存容量的增加和虚拟储存器的使用，编程人员自然希望能寻址更大的范围。所有这些事情（操作码、操作数、寻址方式和地址范围）都需要更多指令的位，并使指令长度趋向更长的方向。但过长的指定长度将是浪费。一个64位指令占据32位指令的两倍空间，但很有可能功能没有两倍那样多。

除了这个基本权衡之外，还有其他的考虑。指令长度或者应该等于存储器的传送长度（在总线系统中，是数据总线宽度），或者这两个值其中一个是另一个的整数倍，否则，会在取指周期得不到整齐数目的指令。一个相关考虑是存储器传送速度。这个速度的提高并不与处理器速度提高保持一致，于是，若处理器执行指令的速度快于取指令的速度，则存储器传送就变成一个瓶颈。这个问题的一种解决办法是使用cache（参见4.3节）。另一种办法是使用较短的指令。这样，16位指令能以32位指令的两倍速率来取指，但执行速度可能不会是两倍那样快。

一个看起来有些平凡的但依然重要的特征是，指令长度应当是字符长度（通常是8位）或定点数长度的整数倍。为表明这一特征，人们使用了一个不幸被含混定义的名词“字”（Word）[FRAI83]。在某种意义上，字的长度是存储器组织的“自然”单位。字的长度通常也就确定了定点数的长度（一般二者是相等的）。字的长度一般也是等于，或至少整齐地相对于存储器传送的宽度。因为数据的普遍形式是字符数据，我们希望一个字能保存整数倍的字符。否则，当存储多个字符时每个字会有浪费的位，或者一个字符必须跨越字的边界。说明其重要性的一个例子是，当IBM推出System/360并打算使用8位字符时，它毅然做出转向的决定，由700/7000系列的36位的科学计算体系结构改为32位结构。

13.3.2 位的分配

我们已经讨论了影响指令格式长度确定的某些因素。一个同样困难的设计出发点是如何分配指令中的位，这个问题的权衡考虑是复杂的。

对于一个给定的指令长度，显然在操作码数目和寻址能力之间有一个权衡考虑的问题，越多的操作码明显地意味着操作码字段要有更多的位，这就减少了寻址可用的位数。对于这种折中考虑有一个有趣的改进，是使用变长的操作码。按照这种方法，有一个最小操作码长度，但是对于某些操作码，可通过使用指令附加位的方法来指定附加的操作。对于一个固定长度的指令来说，就使留给寻址使用的位减少了，于是，这种方法只适用于要求较少操作数和（或）不太强的寻址方式的指令。

下面一些相互关联的因素，在确定如何使用寻址位时是需要考虑的。

\* **寻址方式的数目：**有时，一种寻址方式能隐含确定，例如，某些操作码会总是使用变址。在其他情况下，寻址方式必须是显示指定的，这将需要一位或多位的寻址方式位

\* **操作数数目：**我们已看到过（例如，图12-3），操作数少会使程序变长而且难于编写。当今的机器上的典型指令都提供两个操作数，每个操作数可能都要求有自己的寻址方式指示位，或者限制只允许两个操作数其中一个使用寻址方式指示位。

\* **寄存器与存储器比较：**一个机器必须有寄存器，这样数据才能装入到CPU进行处理。对于只有一个单一的用户可见的寄存器（通常叫做累加器），一个操作数的地址是隐含的因而不占用指令位。然而，单一寄存器的编程很棘手，而且要求较多的指令。尽管是有多个寄存器，也只需要较少的位来指定寄存器。能用于操作数引用的寄存器越多，指令需要的位数越少。多项研究结果都指出，总计8到32个用户可见的寄存器是比较合适的[LUND77, HUCK83]。大多数当代处理器体系结构至少有32个寄存器。

**\* 寄存器组的数目：**大多数当代机器只有一组通用寄存器，通常是32个或更多寄存器，这些寄存器既能用于保存数据，也能用于保存偏移寻址方式的地址。包括x86在内的一些体系结构具有两个或多个专用寄存器组（像数据和偏移量）。这种方法的优点之一是，对于固定数目的寄存器，功能上的分开将使指令只需较少的位数。例如，有两个8寄存器的组，只需3位来标识一个寄存器；操作码将隐式地确定哪一组寄存器被引用。

**\* 地址范围：**对于引用内存位置的地址来说，地址范围与指令的地址位数有关。因为指令中地址位数严重受到限制，所以很少使用直接寻址。使用偏移寻址，范围问题就出现在地址寄存器长度上。尽管如此，允许对地址寄存器做相当大的偏移，在应用中还是很方便的，这就要求指令中有比较多的地址位数。

\* **地址粒度：**对于引用到存储器而不是寄存器的地址，另一考虑因素是寻址的粒度。在一个有16位或32位字的系统中，一个地址是能引用到一个字还是一个字节，具体由设计者选择。字节寻址对于字符处理是很方便的，但对于一个固定大小的存储器来说却要求更多地址位。

于是，设计人员面对许多因素需要考虑和权衡。然而，各种选择的重要程度却不是很清楚。作为一个例子，我们援引一份研究报告[CRAG79]，他比较了各种指令格式的构成方法，包括使用栈、通用寄存器、一个累加器和仅存储器到寄存器方法。在一致的一组假设条件下，观察到在代码空间或执行时间方面并没有显著的不同。

让我们简要地查看两类机器是如何权衡这些不同因素的。

1、PDP-8

对于通用计算机，最简单的指令设计实例之一是PDP-8机[BELL78b]，它使用12位指令和12位的字，有一个单一的通用寄存器，即累加器。

尽管有这种设计限制，但它的寻址还是非常灵活的。每个存储器引用由7位加上两个1位的修饰符（modifier）组成。存储器分成固定长度的页，每页有27=128字。地址计算是基于对页0或当前页（含有这条指令的页）的引用。两个修饰符其中一个是页位，他确定是引用到页0还是当前页。第2个修饰符指示是直接地址还是间接地址。这两种方式能组合使用，故一个间接地址可以是第0页或者当前页中的一个字所容纳的12位地址。另外，第0页上的8个专用字是自动变址的“寄存器”。当对这些位置上的某一个进行间接访问时，就相当于前变址（preindexing）。

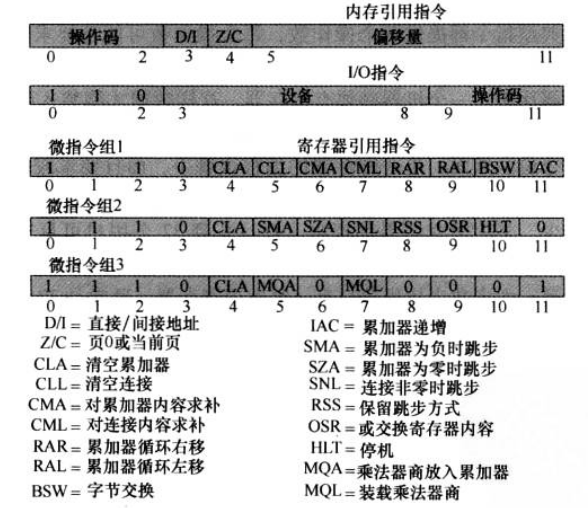
 图13-5表示PDP-8的指令格式。它有3位操作码和3类指令。对于操作码值0~5，指令格式是一种单地址存储器引用指令，带有一个页位和一个间接位。于是，这种指定格式仅有6种基本操作。为扩大操作种类，操作码7定义了一种寄存器引用指令或称微指令（microinstruction）。以这种格式，其余位用于编码其他的操作。通常是，每位定义一种专门操作（例如，清除累加器），而且这些位能组合在一条指令中。采用微指令的设计思路可回溯到DEC的PDP-1。在某种意义上，它是当今微程序式机器的先驱，这种机器将在第四部分讨论。操作码6是I/O指令；6位用于选择64个设备中的某一个，3位用于指定一个具体的I/O命令。

图13-5 PDP-8指令格式

PDP-8指令格式是非常有效的，它支持间接寻址、偏移方式和变址，包括对操作码扩展的操作在内，它能支持约35种指令。对于给定的12位指定长度的限制来说，设计人员已做的很不错了。

2、PDP-10

PDP-8指令集的一个鲜明对照是PDP-10指令集，PDP-10的设计目的是为大型分时系统所使用，并强调编程的易用性，尽管为此会付出更多的硬件代价。

设计该指令集时所采用的设计原则主要有[BELL78c]:

\* **正交性（orthogonality）:**它是指两个变量相互独立的一种原理。在指令集语境中，此术语指的是指令中其他元素独立于操作码，即不被操作码所确定。PDP-10设计者使用这个术语来描述这样一个事实：地址总是以独立的方式来计算，与操作码无关。与之对照的是，不少机器的寻址模式有时是隐含地取决于操作码。

× **完整性（completeness）：**每种算术数据类型（整数、浮点数、实数）都应有一组完整的和等效的操作。

× **直接寻址（direct addressing）：**提倡直接寻址方式，避免使用基址加偏移量寻址方式，因为这种方式把存储器组织的负担放到了程序员身上。

这些原则的每一个都是为了提高编程易用性这一主要目标。

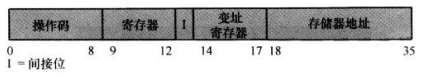


Figure 2的深V

Figure 3

图11-6 PDP-10指令格式

PDP-10有36位字长和36位指令长度，图13-6给出了固定的指令格式。操作码占据9位，允许多达512种操作，实际上PDP-10定义了总共365种指令。大多数指令有双地址，其中一个是16个通用寄存器中的某一个。于是，这个寄存器操作数的引用占据了4位，另一个操作数的引用包含了一个18位的存储器地址字段。它既可用作立即数也可用作存储器地址。在后一种方案中，允许变址和间接寻址，变址寄存器使用的就是第一个操作数引用的同一通用寄存器。

36位的指令长度确实过于奢侈。不管需要这么多操作码做什么，9位操作码字段也显得太多。寻址是直截了当的。18位地址字段使直接寻址更为合理。对于大于218次方的存储器容量，提供了间接寻址。为使编程容易，还为表处理和迭代程序提供了变址。另外，18位的操作数字段使立即寻址变得有吸引力。 PDP-10指令集设计实现了前面所列的目标[LUND77]，它以空间利用率低为代价，使得程序员编程相对容易。这是设计人员有意而为之，不能把它错看成一种糟糕的设计。

13.3.3 变长指令

至此，我们已考察了单一固定指令长度的使用，并在上下文中间隐含的讨论了所做的权衡考虑。但设计者可选取另一种替代方法，即提供不同长度的各种指令格式。这种策略易于提供大的操作码清单，而操作码具有不同的长度。寻址方式也能更灵活，指令格式能将各种寄存器和存储器引用加上寻址方式予以组合使用。使用变长指令，能有效和紧凑地提供这些众多变化。

变长指令的主要代价是增加了CPU的复杂程度。硬件价格的降低，微程序设计方式的使用（第四部分将讨论）以及对CPU设计原则理解的普遍提高，这一切都使所付的代价变小。但是，我们会看到RISC和超标量机器能利用固定长度的指令来提高性能。

使用变长指令并没有消除所有指令相对于机器字长，其长度整齐的期望。因为CPU不知道下一条待取指令的长度，典型的策略是取至少等于最长指定长度的几个字节或几个字。这意味着有时一次能取来多条指令。不过，我们会在第14章看到，在任何情况下，这都是一个好的策略。

1. PDP-11

PDP-11设计是在16位小型计算机范畴内提供了功能最强和最灵活的指令集[BELL70].

PDP-11使用了一组8个16位通用寄存器。其中两个有特殊的作用，一个用作栈指针，支持专用的栈操作。一个用作程序计数器，保存下一条指令的地址。

图13-7给出了PDP-11的指令格式。PDP-11使用了13种不同格式，包括零地址、单地址和双地址指令类型。操作码的长度在4位到16位。寄存器引用使用6位，其中3位用于指定寄存器，其余3位用于指定寻址方式。PDP-11具有丰富的寻址方式。将寻址方式关联到操作数而不是操作码，其优点是任何寻址方式能与任何操作码一起使用。正如前面提到的，这种独立性称为正交性。

PDP-11指令通常是1字（16位）长。对于某些指令，又续加了一个或两个存储器地址，这样32位和48位指令也成为指定清单的一部分。这进一步提供了寻址的灵活性。

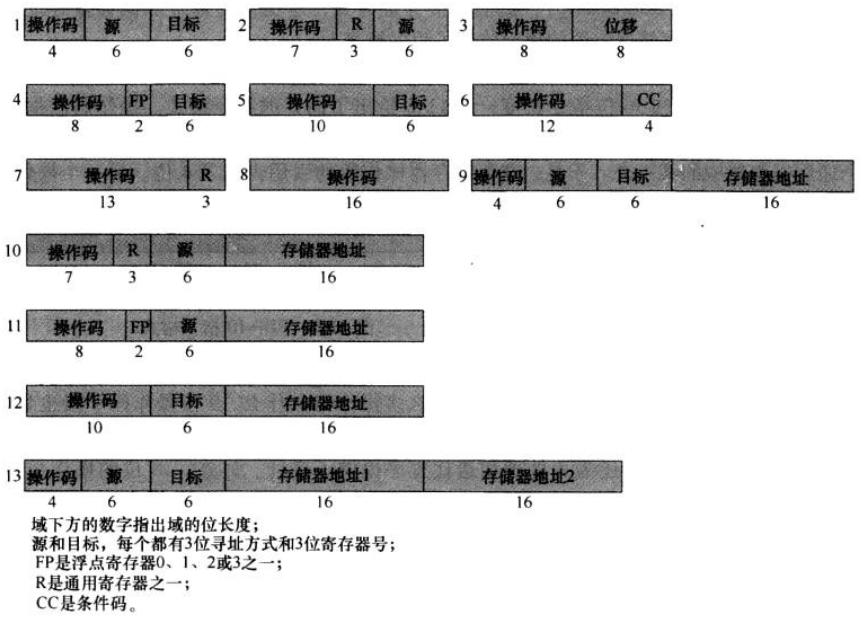
 PDP-11指令集和寻址能力是复杂的，这增加了硬件成本和编程的复杂性。有优点是能开发更有效更紧凑的程序。

图 13-7 PDP-11指令格式

2. VAX

大多数处理器体系结构提供相对比较少的固定指令格式。这对程序设计人员来说，导致了两个问题。首先，寻址方式和操作码是非正交的。例如，对某个操作，其操作数必须来自寄存器，另一个来自存储器，或者两个都来自寄存器等等。其次，一条指令只能使用有限数目的操作数，一般最多只有2~3个。由于某些运算本身就要求多个操作数，这就要使用某种技巧以两条或更多条指令来实现所要求的运算。 为避免这些问题，设计VAX指令格式时确定了两条规则[STRE78]:

（1）所有指令都应该具有“天生”数目的操作数。

（2）所有操作数都应该具有同样的操作通则。

结果是高度可变的指令格式。一条指令由1或2字节的操作码后跟随0~6个操作数规定符（specifier）组成，规定符的具体数目取决于操作码。最短的指令是1字节长，但也能有多达37字节的指令。图13-8给出了VAX指令的若干例子。

VAX指令以1字节操作码开始。对于大多数指令，1字节操作码足够了，然而VAX指令有300多条。于是，少数指令要使用2字节操作码：第一字节为FD或FF，指示第2字节为实际操作码。

操作码之后是可多达6个的操作数规定符。最小操作数规定符是1字节，其格式最左4位用于寻址方式说明。唯一例外是立即方式，此时第1字节最左两位是00，留出6位用于指定立即数。由于这个例外，4位字段总计能说明12种寻址方式。



图 13-8 VAX指令举例

操作数规定符经常只由1个字节组成，此时最右4位用于指定16个通用寄存器之一。操作数规定符能以如下两种方式之一来扩展长度。第一种方式是，一个或多个字节的常数值可紧跟操作数规定符第1个字节之后。例如，偏移寻址方式中的8位、16位或32位的偏移量。第二种方式是用变址方式，此时第1字节由0100变址方式码和4位变址寄存器标识符组成，操作数规定符的其余字节用于指明基地址。

读者可能会感到惊讶，什么指令需要6个操作数，而VAX还确实有这样的指令，例如：

ADDP6 OP1, OP2, OP3, OP4, OP5, OP6

这条指令将两个压缩十进制数相加，OP1和OP2指明一个十进制串的长度和起始地址，OP3和OP4指明另一个串，相加结果串的长度和存放始地址由OP5和OP6指示。

VAX指令集有范围广泛的操作类型和寻址方式，这为程序设计人员，尤其是编写编译程序的程序员提供了一种强有力的、灵活的编程工具。从理论上讲，这将有利于高效地把高级语言程序编译为机器语言，以及对CPU资源的有效利用。但为此付出的代价是，与具有简单指令集和格式的处理器相比，VAX CPU的复杂性大幅度地增加了。

第15章考察精简指令集的情况，届时我们将继续讨论这个问题。

13.4 x86和ARM指令格式

13.4.1 x86指令格式

x86配备了各种指令格式。下面介绍的指令各元素中，只有操作码字段是必出现的，其他都是可选的。图13-9说明了通常的指令格式，。指令由0到4个字节的可选指令前缀、1或2字节的操作码、一个可选地址指定符（由Mod R/m字节和比例变址(Scale index)字节组成）、一个可选的偏移量以及一个可选的立即数字段等组成。

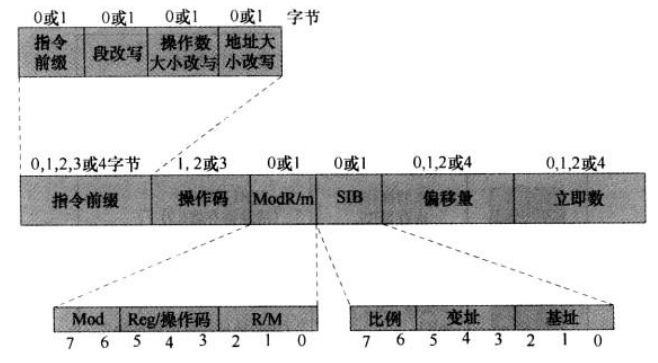


图13-9 x86指令格式

下面首先考察前缀字节。

**× 指令前缀（instruction prefix）：**指令前缀若出现，则由lock（锁定）前缀或一个重复前缀所组成。lock前缀用于多处理器环境，保证对共享存储器的独占式访问。重复前缀指定串的重复操作，这就使x86处理器串要比普通的软件循环快得多。有5种重复前缀：REP、REPE、REPZ、REPNE和REPNZ。当无条件的REP前缀出现时，指令中指定的操作对串中连续元素重复执行，重复的次数由CX寄存器指定。条件REP前缀出现时，引起指令重复执行直到CX变为0或指定的条件被满足。

× **段取代**（segment override）：显式的指定这条指令应使用哪个段寄存器，取代x86为那条指令规定的默认段寄存器。

× **操作数大小**（operand size）：指令默认的操作数大小是16位或32位，操作数大小前缀用于在32位和16位的操作数大小之间的切换。

× **地址大小**（address size）：处理器能使用16位或32位地址来寻址寄存器。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。

指令本身包括如下字段

× **操作码**：操作码字段长度是1、2或3字节。操作码可能包括一些位，这些位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（送至和来自存储器），以及一个立即数字段是否要进行符号扩展。

× **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使用的寻址方式。Mod R/m字节由三个字段组成：Mod字段（2位），与R/m字段组合构成32个可能的值：8个寄存器和24个变址方式。Reg/操作码字段（3位）能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字段组合成编码。

**× SIB**：Mod R/m字节的某些编码要求包含另一个称为SIB字节来完成寻址方式的指定。SIB字节由三个字段组成：比例（scale）字段（2位）指定用于比例变址的比例因子；变址（index）字段（3位）用于指定变址寄存器；基址（base）字段（3位）用于指定基址寄存器。

×**偏移量**：当地址方式指定符指出使用一个偏移量时，一个8位、16位或32位有符号整数的偏移量被添加到指令中。

× **立即数**：在指令中提供一个8位、16位或32位的操作数值。

在这里做几个比较可能会更有助于理解。在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。因为只允许一个操作数有寻址方式信息，所以，x86指令中也就只能引用到一个存储器操作数。相对比，VAX机是每个操作数都可携带寻址方式信息，从而允许存储器到存储器的操作。因此，x86的指令更紧凑。然而，若要求存储器到存储器的操作，VAX使用一条指令就能实现。

x86格式允许变址使用不仅是1字节，而且是2字节或4字节的位移。虽然使用较长的变址位移会导致指令变长，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。与之对比，IBM S/370指令格式只允许位移不大于4kb（12位的位移信息），并且位移必须是正值。当一位置不在此位移范围内时，编译器必须生成额外的代码来产生所需的地址。当与局部变量超过4kb的栈帧打交道时，这个问题变得尤为明显。正如[DEWA90]对它的描述：“由于那个限制，为370生成代码非常费劲，导致有的370编译器简单地选择把栈帧的大小限定到4kb。”

正如我们已看到的，x86指令集的编码是很复杂的。其部分原因是与8086向下兼容的需要，部分原因是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，是像这样的复杂指令集还是另一极端的RISC指令集更合适一些，还是一个有争议的事情。

13.4.2 ARM指令格式

 ARM的所有指令都是32位长，并有规整的格式（见图13-10）。指令的前4位是条件码。正如在第十章中讨论的，实际上ARM的所有指令都是条件执行的。指令接下来的3位指定了指令的一般类型。对除分支指令之外的大多数指令而言，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指令这种规整的格式使得指令译码单元的工作变得比较轻松。

图13-10 ARM指令格式

1、立即常数

为获得取值范围较大的立即数，采用立即数的数据处理指令不但指定了立即数值，还指定了一个循环移位值。8位的立即数值被扩展到32位，然后循环右移若干次，次数等于4位循环移位值的两倍。图13-11显示了几个这方面的例子。



图13-11 使用ARM立即常数的例子

2、压缩指令集

压缩指令集（thumb instruction set）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实现的性能，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。这样总计省去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，从而总计省去了16位。具体来说，例如压缩指令只引用此寄存器r0到r7，因此只需要3位的寄存器引用字段，而不是4位。立即数字段中也省去了4位的循环移位量字段。

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。图13-12给出了这样的例子。图中给出两种类型指令的一般格式，以及16位和32位指令的具体示例。

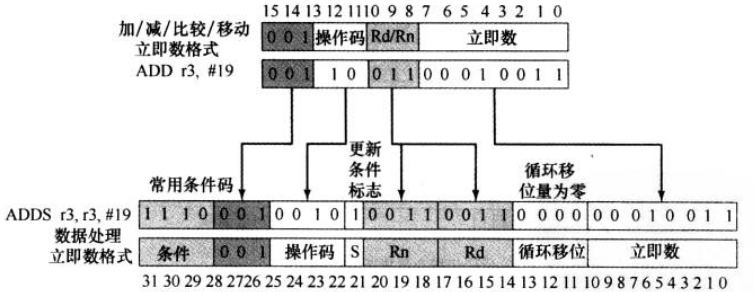


图13-12 将一条压缩ADD指令扩展为对应的普遍ARM指令

3、？？指令集

正如上述对压缩指令集的介绍，用户需要通过将执行代码编译为ARM指令和压缩指令来混合指令集。这种手动混合指令的方法需要更多的工作量，却很难得到理想的效果。为了解决这些问题，ARM使用了Thumb-2指令集，它这是Cortex-M微处理器上唯一可用的指令集

Thumb-2指令集是对压缩指令架构（ISA）的增强版本。它讲解了可以自由混合以前的16位混合指令集的32位指令。这种新的32位指令涵盖了几乎所有的ARM指令集的功能特性。压缩指令集架构和ARM指令集架构最重要的不同在于大多数32位压缩指令集没有使用条件，而ARM指令是有使用条件的。thumb-2指令介绍了一种新的If-Then（IT）指令，它提供了ARM指令中条件字段的大部分功能。

Thumb-2提供了与压缩指令集相当的代码密度和与ARM指令相当的执行性能。在没有thumb-2指令集之前，开发者必须选择使用压缩指令集以追求代码精简程度，或者选择使用ARM指令集一追求执行性能。

[ROBI07]阐述了关于对Thumb-2指令集与ARM指令集和压缩指令集的比较分析报告。这份报告通过使用三种指令集来编译和执行嵌入式微处理器基准组合(EEMBC)基准套件来进行分析，结果如下：

× 在最优化性能的情况下，thumb-2代码量比ARM代码小了26%，略大于压缩指令编写的代码。

× 在最优化代码精简度的情况下，thumb-2代码量比ARM代码小了32%，略小于压缩指令编写的代码。

× 在最优化性能的情况下，在基准套件测试中thumb-2代码性能是ARM指令集代码的98%，是压缩指令集代码的125%。

这些结果证明thumb-2满足设计目标。

图13-13表示了新版32位压缩指令集是如何编码的。编码与现有的压缩指令中的无条件分支指令（在指令的最左边5位具有11100的位模式）兼容。16位指令中，最左3位不会以111的为模式开始，所以位模式11101,11110和11111隐含意思就是这是一个32位压缩指令集。

11.5 汇编语言

处理器可以识别和执行机器指令。这些指令都是以二进制形式存储在计算机中的。如果一个程序员想直接用机器语言编程，那么它必须用二进制来编写程序。

考虑下面这个简单的BASIC语句：

N=I+J+K

假设我们想用机器语言来编写这个语句，并且把I 、J和K分别初始化为2，3和4。图13-14a给出了机器语言的程序。程序从内存位置101（十六进制）开始。所用到的4个变量在内存中保留在从位置201开始的地方。程序包括下面这4条指令：

（1）把位置201的内容装载到累加器AC中。

（2）把位置202的内容加到累加器AC中。

（3）把位置203的内容加到累加器AC中。

（4）把累加器AC中的内容保存到位置204中。

这样编程很明显十分繁琐，而且容易出错。

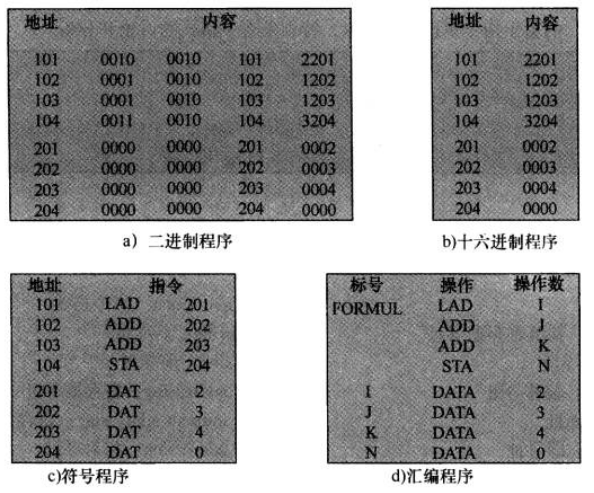
一个稍微的改进是用十六进制，而不是用二进制来编写程序（图13-14b）。我们可以把程序写成一系列的行。每行对应着一个内存位置的地址，以及要保存到这个位置的二进制内容的十六进制表示。然后我们需要一个程序接收这些行作为输入，把每行的十六进制数转换为二进制数，并把它们保存到所指定的位置。

图13-14 公式N=I+J+K的计算

更多的改进是，我们可以利用符号名称或助记符来表示每条指令。这就导致了图13-14c所示的符号程序。输入的每一行仍然对应着一个内存位置。每行包括三个字段，以空格隔开。第一个字段是该内存位置的地址。第二个字段是每条指令操作码对应的三个字母的符号。如果一条指令是内存引用指令，那么第三个字段就包含所引用的内存。要保存任意的数据到某个位置，我们发明了一条伪指令（pseudoinstruction），其符号是DAT。该伪指令只是用来指明该行第三字段的十六进制数要被保存到该行第一个字段所指定的内存位置。

对于这种类型的输入程序，我们需要一个稍微复杂一些的程序。该程序接收输入的每一行，根据每行第二字段和第三字段（如果有的话）生成相应的二进制数，并保存到第一字段指定的地址中。

使用符号程序可使编程工作轻松些，但还是比较笨拙。尤其是，必须为每个字指定一个绝对地址时。这意味着程序指令和数据都只能被装载到内存中一个固定的地方，而且还必须预先确定。更糟糕的是，假如我们某天想修改程序，添加或删除一行，那么我们必须更新被修改处之后所有行的地址。

更好的一种方式也是普遍使用的一种方式，是使用符号地址，如图13-14d所示。每行还是包含3个字段。第一个字段仍然是地址，但是不再使用绝对的数值地址，而是使用符号。有些行没有地址，意味着这些行的地址是按顺序接着上一行的地址递增。对于引用内存的指令，第三个字段也使用符号地址。 经过最后的改进，我们得到了所谓的汇编语言（assembly language）。用汇编语言编写的程序（汇编程序）通过汇编器（assember）翻译为机器语言。汇编器不但进行上面介绍过的符号操作码翻译而且对符号地址进行存储器地址分配。

汇编语言的出现是计算机技术发展的一个重要里程碑。它迈出了走向今天我们所使用的高级语言的第一步。虽然现在很少有程序员使用汇编语言了，但所有的机器都提供了对应的汇编语言。对于系统程序，例如编译器和输入/输出历程，它们还是需要的。

附录B提供了对汇编语言的更详细的考察。

13.6 关键词、思考题和习题

关键词

|  |  |
| --- | --- |
| Autoindexing：自动变址  Base-register addressing：基址寄存器寻址  Direct addressing：直接寻址  Displacement addressing：偏移寻址  Effective addressing：有效地址  Immediate addressing：立即寻址  Indexing：变址  Indirect addressing：间接寻址 | Instruction format：指令格式  Postindexing：后变址  Preindexing：前变址  Register addressing：寄存器寻址  Register indirect addressing：寄存器间接寻址  Relative addressing：相对寻址  Word：字 |

思考题

13.1 立即寻址有什么优点？

13.2 直接寻址有哪些局限？

13.3 多级间接寻址的缺点是什么？

13.4 简述寄存器寻址的定义。

13.5 简述寄存器间接寻址的定义。

13.6 简述偏移寻址的定义。

13.7 简述相对寻址的定义。

13.8 自动变址有什么优点。

13.9 后变址和前变址有什么不同？

13.10 确定寻址位的使用受什么因素影响？

13.11 使用变长指令的优缺点是什么？

习题

13.1 给定如下存储器的值，并使用有一累加器的单地址机器，下列指令将把什么值装入累机器？

\* 字3含15

\* 字12含18

\* 字15含24

\* 字18含36

A、LOAD IMMEDIATE 3

B、LOAD DIRECT 3

C、LOAD INDIRECT 3

D、LOAD IMMEDIATE 15

E、LOAD INDIRECT 12

F、LOAD DIRECT 18

13.2 若存于程序计数器的地址标记为X1，存于X1中的指令的地址部分（操作数引用）是X2，执行此指令所需的操作数存于地址为X3的存储器字中。变址寄存器有值X4。若此指令的寻址方式是（a）直接寻址，（b）间接寻址，（c）PC相对寻址，（d）变址寻址，那么X1，X2，X3，X4应该如何组合，从而得到需要的地址？

13.3 某指令的地址字段含有十进制值96，对于下列寻址方式，其对应的操作数位于何处？

（a）立即寻址 （b）直接寻址 （c）间接寻址

（d）寄存器寻址 （e）寄存器间接寻址

13.4 考虑一个十六位处理器，它的一条装载指令以如下形式出现在主存，起始地址为100。第一字的第一部分指出此指令是将一个值装入累加器。Mode字段用于指定一种寻址方式。对于某些寻址方式，Mode字段还指定了一个源寄存器，并假设指定的源寄存器是R1，其中的值为200，还有一个基址寄存器，其值为100。在地址是101处的值300可以是地址计算的一部分。假定地址199处有值799，地址200处有值800……请对如下寻址方式确定有效地址和被装入的操作数：

（a）直接寻址 （b）PC相对寻址 （c）寄存器间接寻址

（d）立即寻址 （e）偏移寻址 （f）使用R1自动递增寻址

（g）间接寻址 （f）寄存器寻址

13.5 某PC相对方式的分支指令是3字节长。此指令的地址是256028（十进制）。若指令中的带符号偏移量是-31，请确定转移的目标地址。

13.6 某条PC相对方式的分支指令存于地址602的存储器位置中，它要转移到50310位置上。指令中的地址字段是是10位长，其二进制值是什么？

13.7 若CPU取并执行一条间接地址方式指令，该指令是：（a）一个要求单操作数的计算；（b）一个分支，那么CPU分别需要访问存储器几次？

13.8 IBM 370不提供间接寻址。假定一个操作数的地址是在主存中，你如何存取此操作数？

13.9 在[COOK82]中作者建议取消PC相对寻址方式，赞成使用其他寻址方式，例如栈寻址方式，这个建议有什么缺点？

13.10 X86包括如下指令：

IMUL op1, op2, immediate

这条指令将操作数p2（可以是寄存器或存储器）乘以立即操作数值，结果放入op1（必须是寄存器）中。指定集中在没有其他这类的三操作数指令，这种指令的可能用途是什么（提示：考虑变址）？

13.11 考虑一个包括基址带变址（base with indexing）寻址方式的处理器。假设遇到使用这种寻址方式的一条指令，指令给定的偏移量是1970（十进制）。当前的基址和变址寄存器分别有十进制数48022和8。那么操作数的地址是什么？

13.12 定义：EA=（X）+ 为有效地址等于位置X的内容，并在有效地址计算后X递增一字长；EA= -（X）为有效地址等于位置X的内容，并在有效地址计算前X递减一字长；EA=（X）- 为有效地址等于位置X的内容，并在有效地址计算后X递减一字长。考虑如下指令，它们都有（操作、源操作数、目的操作数）的格式，并且操作结果放入目的操作数。

（a）OP X，（X） （b）OP （X），（X）+

（c）OP （X）+，（X） （d）OP -（X），（X）+

（e）OP -（X），（X）+ （f）OP （X）+，（X）+

（g）OP （X）-，（X）

使用X作为栈指针，上述那些指令能由栈弹出顶部两元素，完成所需要的操作（例如，ADD源到目的并存入目的），并将结果压回栈？对这样的每条指令，栈是朝存储器位置0方向还是朝相反方向增长？

13.13 假定有一个面向栈的处理器，包括有PUSH和POP栈操作。算术运算自动涉及栈顶的1或2个元素。开始时栈为空。下述指令执行后栈中保留下来的栈元素是什么？

PUSH 4

PUSH 7

PUSH 8

ADD

PUSH 10

SUB

MUL

13.14 证明说法：32位指令的功能不会有16位指令的功能两倍那样多，是正确的。

13.15 为什么IBM决定将每字36位转向到每字32位的结构，并且是针对什么做出的这一决定？

13.16 假定有一指令集，其指令长度是固定的16位长，其中操作数指定符是6位长。若有K条双操作数指令，L条零操作数指令，那么能支持的单操作数指令的最大数目为多少？

13.17 设计一种变长操作码，以允许如下指令全部都能编码成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

\* 指令没有地址和寄存器号

13.18 考虑习题10.6的结果，假定M是一个16位存储器地址，X、Y、Z或是16位地址，或是4位寄存器号。单地址机器使用一个累加器。双地址和三地址机器有16个寄存器，并且指令能在存储器位置和寄存器的各种组合上操作。并假定指令长度是4位的整数倍，操作码是8位。为计算X，每种机器各需要多少位？

13.19 一条指令有两个操作码，它有无任何可能的存在理由？

13.20 16位的Zilog Z8001通常有如下指令格式：

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 模式 | | 操作码 | | | | | w/b | 操作数2 | | | | 操作数1 | | | |

其中模式字段指示如何由操作数字段找到操作数。w/b字段用在某些指令中指示操作数是字节（B）还是16位的字（W）。操作数1字段可以（取决于模式字段内容）指定16个通用寄存器之一。操作数2字段可指定除寄存器0之外的任一通用寄存器。若此字段为全0，则原操作码有新的意义。

1. Z8001提供了多少的操作码？
2. 建议一种有效方式来提供更多操作码并指出所涉及的权衡考虑。