第3章 汇编级机器组织

现代计算机是按人们预先编制的程序进行工作的，而程序是由指令组成的。计算机能直接识别、执行的命令称为机器指令(简称指令)，机器指令用二进制编码表示。一条指令规定计算机完成某种基本操作，一台计算机能执行的所有指令的集合称为该计算机的指令系统(或称指令集)。

指令系统是计算机系统性能的集中体现，其功能与格式不仅直接影响到计算机硬件结构的设计，而且与系统软件的设计、计算机的应用领域等息息相关。通常性能较好的计算机都设置有功能齐全、通用性强、指令丰富的指令系统，而指令功能的实现又需要复杂的硬件结构来支持。

虽然机器指令能被计算机直接识别和执行，但是难于书写、记忆，出错不易发现。汇编指令是机器指令的助记符表示形式，与机器指令是一一对应的。基于此，本章将讨论计算机中汇编级指令的格式、地址结构，指令及操作数的寻址方式，指令的种类和功能，指令集的架构和典型的指令系统举例等内容。

3.1 汇编级机器指令系统

计算机在汇编级机器指令系统的设计上，既要考虑指令系统功能的需要，又要考虑实现其所需机器硬件的成本。但不管怎样，对机器的指令系统和指令种类都有一些基本的要求。

3.1.1 指令系统的发展

在现代计算机的发展过程中，计算机的设计、编程语言、制造工艺技术都发生了深刻的变革。伴随着这种变革，计算机指令系统的发展经历了从简单到复杂的演变过程。

20 世纪 50—60 年代是现代计算机发展的早期，采用电子管或晶体管等分立元件组成计算机硬件，其体积庞大，价格也很昂贵，因此计算机的硬件结构比较简单，所支持的指令系统也只有定点加减、逻辑运算、数据传送、转移等十几至几十条最基本的指令，而且寻址方式简单。

到了 20 世纪 60 年代中期，随着集成电路的出现，计算机的体积减小，功耗降低，价格不断下降，硬件功能不断增强，指令系统也越来越丰富。除最基本的指令外，增加了乘除运算、浮点运算、十进制运算、字符串处理等指令，指令数目有一二百条，寻址方式也趋于多样化。

随着集成电路的发展和计算机应用领域的不断扩大，20 世纪 60 年代后期开始出现系

列计算机。系列计算机是指基本指令系统相同、基本体系结构相同的一系列计算机，例如Pentium系列微型计算机。一个系列往往有多种型号，但由于推出的时间不同，采用的器件不同，它们在结构和性能上有所差异。通常是新推出机种的指令系统一定包含所有旧机种的全部指令，旧机种上运行的各种软件可以不加任何修改便可在新机种上运行，大大减少了软件开发费用。系列计算机不仅较好地解决了各机种的软件兼容问题，而且新机种的性能价格比优于旧机种。

20世纪70年代，高级语言已成为大、中、小型机的主要程序设计语言，计算机应用日益普及。但是复杂的软件系统设计一直没有很好的理论指导，导致软件质量无法保证，从而出现了所谓的“软件危机”。人们认为，缩小机器指令系统与高级语言语义的差距，为高级语言提供更多的支持，是缓解软件危机有效和可行的办法。计算机设计者们利用当时已经成熟的微程序技术和飞速发展的超大规模集成电路技术，增设了多种复杂的、面向高级语言的指令，使指令系统越来越复杂化，大多数计算机的指令系统多达几百条，由此就出现了复杂指令系统计算机(complex instruction set computer，CISC)。

但CISC庞大的指令系统不但使计算机的研制周期变长，正确性难以保证，不易调试和维护，而且由于大量使用频率低的指令而造成硬件资源浪费。因此，计算机设计者尝试从另一条途径来支持高级语言及适应VLSI的技术特点。1975年，IBM公司的John Cocke提出了精简指令系统计算机(Reduced Instruction Set Computer，RISC)的设想。有关CISC和RISC的内容将在后续的3.4节详细介绍。

20世纪80年代后期以来，RISC微处理器迅速发展，广泛采用了超标量和超流水线等指令级并行处理技术。但是人们又发现RISC指令系统并不能充分实现指令级并行处理，从而影响了计算机性能的进一步提高。1983年，美国教授J. Fisher提出了超长指令字(very long instruction word，VLIW)体系结构。VLIW指令系统的设计思想是增强指令的并行性，使机器指令字具有固定的格式(一种或者多种)，每条指令中包含着多个独立的字段，字段中的操作码被送往不同的功能部件。这是受微程序设计技术中水平型微指令的启示。

总之，计算机指令系统的发展经历了从简单到复杂，然后又从复杂到简单的演变过程。

3.1.2　指令系统性能的要求

指令系统的性能决定了计算机的基本功能，它的设计直接关系到计算机的硬件结构和用户的需要。由于计算机性能、体系结构以及使用环境等要求不同，指令系统间的差异也是很大的。同时，随着计算机技术的迅速发展，对计算机性能的要求越来越高，硬件价格迅速下降，指令系统也趋向强功能、高效、复杂化。因此，试图给计算机指令系统确定一个统一的衡量标准是很困难的。一般情况下，指令系统应满足完备性、有效性、规整性和兼容性。

1. 指令系统的完备性

完备性是指在一个有限可用的存储空间内，对于任何可解的问题，在编写计算程序时，指令系统所提供的指令应足够使用。一般来说，为使程序高效运行和便于硬件实现，一个完备的指令系统应包括传送类指令、算术运算类指令、逻辑运算类指令、程序控制类指令、字符串指令、特权指令和调试指令。若采用统一编址，则用传送类指令即可实现输入输出的目

的；若采用单独编址，则需要专门的输入输出类指令。

以上几种类型的指令并不是互相独立的，在一定条件下可以互相替代。

2. 指令系统的有效性

有效性是指使用该指令系统编制的程序能高效率地运行。这个高效率表现在解题速度快，占用存储空间小。有效性是针对整个指令系统而言的，这是一个很复杂的问题，很难确定一个统一标准。有效性反映了指令的功能要求，也反映了指令系统的完备性要求。一个更完备的指令系统就会有更好的有效性，如指令系统中有多种移位指令和字符串处理指令，对于数据处理就会有较高的有效性。新一代计算机指令系统中有专门的编辑指令，甚至某些常用的高级语言的语句也用一条指令来实现等，采取这些措施无疑将大大提高指令系统的有效性。

3. 指令系统的规整性

规整性包括指令操作的对称性和匀齐性以及指令格式与数据格式的一致性。

指令操作的对称性是指在运算时，所有寄存器(存储)单元都可以同等对待，不论哪一个操作数或运算结果都可以不受约束地存入任一单元。如传送指令既有 A←B，也有 B←A；加法指令既有 A←A＋B，也有 B←A＋B 等。这种操作的对称性对于提高软件效率和使用便利性是很有利的，但是，目前许多机器还不能很好地实现这一对称性。相较而言，VAX-11 机指令系统的指令操作具有较好的对称性。

指令操作的匀齐性是指一种性质的操作可适用于各种数据类型。如 VAX-11 机在数据传送、数据交换、数据测试和计算等操作时，可以匀齐地适用于三种整数数据类型(字节、字和双字)和两种浮点数据类型(短浮点和长浮点)。指令操作的匀齐性可使汇编程序设计与编译程序无须依赖数据类型去选用指令，缩短了程序代码长度，加快了程序执行速度。匀齐性在以前的一些机器中没有很好地实现。如 IBM 370 机有字(32 位)加法，也有半字(16 位)加法；但只有字除法，却没有半字除法等。随着硬件价格的进一步下降，指令操作的匀齐性有了很大的改善。

指令格式与数据格式的一致性是指指令字长与数据字长有一个规整的关系，便于程序的加工处理。如机器基本字长为 32 位，长指令选 32 位，短指令选 16 位，在此字长的条件下，可以选择指令的地址格式。

4. 指令系统的兼容性

从计算机的发展过程可以看到，由于组成计算机的基本硬件发展迅速，计算机的更新换代是很快的，这就存在软件如何跟上的问题。大家知道，一台新机器推出交付使用时，仅有少量软件可提交用户，大量软件是不断充实的。尤其是应用软件，有相当一部分是用户在使用机器过程中不断产生的，这就是所谓的第三方软件。为了缓解新机器的推出与原有应用软件能否继续使用之间的矛盾，1964 年，IBM 公司在设计 IBM 360 计算机时所采用的系列机思想较好地解决了这一问题。从此以后，计算机公司生产的同一系列计算机尽管其硬件实现各不相同，但在指令系统、数据格式、I/O 系统等方面均保持相同，因而软件完全兼容(在此基础上产生了兼容机)。当研制该系列计算机的新型号或高档产品时，尽管指令系统

可以有较大地扩充，但仍保留原来的全部指令，保持软件“向上兼容”的特点。

系列机的各型号机器由于推出的时间不同，在结构和性能上有差异，做到所有软件都完全兼容是不可能的，只能做到“向上兼容”或“向前兼容”，即低档机运行的软件可以在高档机运行。

指令系统的兼容性使大量已有软件产品得到继承，保护了用户的前期软件投资，减少了软件的开发费用，同时，新型号机器一出现就具有丰富的软件，有利于打开市场销路。

3.1.3　指令操作的种类

不同类型的计算机所具有的指令类型是多种多样的，其指令的数量与功能、指令格式、寻址方式、数据格式都有差别。即使是一些常用的基本指令，如算术运算指令、逻辑运算指令、转移指令等也是各不相同。但是从指令的操作功能来考虑，一个较完善的指令系统应当包括数据传送类指令、算术运算类指令、逻辑运算类指令、程序控制类指令、输入输出类指令、系统控制类指令等。

1. 数据传送类指令

计算机的操作大部分可以归结为各类信息的传送，因此数据传送类指令是最基本的指令。数据传送类指令主要用于寄存器与寄存器、寄存器与存储单元、存储单元与存储单元之间的数据传送操作。对存储器来说，数据传送包括对数据的读/写操作。传送时，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。因此，计算机中的数据传送实际上是数据复制。

数据传送类指令可以是一个操作数的传送，也可以是一串操作数的传送，这取决于指令的操作要求。例如，Intel 8086 的 MOVS 指令一次可以传送一个字节或字。而加上重复执行前缀(REP)后，一次可以把一批数据从存储器的一个区域传送到另一个区域。

数据传送类指令既可以是单向的，也可以是双向的，即将源操作数与目的操作数(一个字节或一个字)相互交换位置。例如 Intel 8086 中的 XCHG 指令，源操作数与目的操作数可互换位置，源操作数可以在寄存器或主存单元中，目的操作数则只允许在通用寄存器中。

2. 算术运算类指令

算术运算类指令主要用于定点数和浮点数运算。这类运算包括定点数的加、减、乘、除运算，浮点数的加、减、乘、除运算，以及自加 1、自减 1、比较运算等。此外，有些机器还有十进制运算指令。绝大多数算术运算类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，以决定程序的走向。

为了实现高精度的加法(减法)运算(双倍字长或多字长)，低位字(字节)加法运算所产生的进位(或减法运算所产生的借位)都存放在进位标志中；在高位字(字节)加法(减法)运算时，应考虑低位字(字节)的进位(或借位)。因此指令系统中除去普通的加减指令外，一般都设置有带进位加指令和带借位减指令，例如 Intel 8086 中的 ADC、SBB 指令。

3. 逻辑运算类指令

一般计算机都具有逻辑与、或、非、异或、移位等指令，这类指令主要用于无符号数的位操作、代码转换、判断及运算。例如 Intel 8086 中的 AND、OR、NOT、XOR 等指令。

移位指令用来对操作数实现左移、右移和循环移位操作。移位指令可分为算术移位、逻辑移位和循环移位三类，同时它们又分别包括左移和右移两种。例如 Intel 8086 中的 SHL/SHR、SAL/SAR、ROL/ROR、RCL/RCR 指令。

4. 程序控制类指令

程序控制类指令主要用于控制程序的执行顺序，并使程序具有测试、分析与判断的能力，因此，它是指令系统的一个重要组成部分。这类指令主要包括无条件转移指令、条件转移指令、转子程序与返回指令、程序中断指令等。

(1) 无条件转移指令。无条件转移指令用来改变指令的正常执行顺序，不受任何约束地将程序转移到该指令指定的地址去执行。这种操作只影响程序计数器的内容，如 Intel 8086 的 JMP 指令。

(2) 条件转移指令。条件转移指令是指仅当满足指令规定的条件时，才执行转移；否则只相当于一条空操作指令，不改变程序执行顺序。条件转移指令一般常跟在比较或测试指令之后，根据测试条件是否满足来决定是否转移。由于这种指令可使计算机根据条件(输入数据处理或处理结果)来选择程序执行方向，因此条件转移指令使机器具有逻辑判断能力。

一般用来作为转移条件的标志有进位标志(C)、结果为零标志(Z)、结果为负标志(N)和结果溢出标志(V)等，也可以利用这些标志的组合作为转移条件，如 Intel 8086 的 JZ、JC 等指令。

(3) 转子程序与返回指令。通常，子程序是一组可以共享的指令序列，在执行主程序的过程中可被多次调用，甚至可被不同的程序所调用。只要给出子程序的入口地址就能从主程序转入子程序。转子程序操作与转移指令的区别在于：转移指令无须返回，不必保存返回地址；而转子程序指令必须以某种方式保存返回地址，以便子程序执行结束后返回到调用程序的断点，如 Intel 8086 的 CALL、RET 指令。

(4) 程序中断指令。中断是计算机内部突发事件或由 I/O 设备请求而随机产生的。但有些计算机为了方便程序调试或调用某种功能等目的，设置有专门的程序中断指令，如 Intel 8086 的 INT 指令。当程序运行该类指令时，就以中断方式暂停后续指令的执行，将程序断点参数保存，然后转向某地址执行某段程序，实现所期望的功能，如查错、显示结果等。由于这些指令是由软件驱动的，所以又称为软中断。

5. 输入输出类指令

输入输出类指令用来启动外围设备，检查测试外围设备的工作状态，实现外围设备与主机之间或外围设备与外围设备之间的信息交换。不同计算机的输入输出方式差别很大，通常有两种方式：独立编址方式和统一编址方式。

独立编址方式设置有专用的输入输出指令，指令中给出了与存储器地址无关的设备码

(或端口地址),而指令操作码规定本指令要求的输入输出操作,包括输入、输出、启动等,如Intel 8086 的 IN、OUT 等指令。

所谓统一编址,就是把输入输出设备寄存器和主存单元统一编址。在这种方式下,用一般的数据传送指令来实现输入输出操作。一个输入输出设备通常至少有三个寄存器:数据寄存器、命令寄存器和状态寄存器。每个寄存器都可以由分配给它的唯一地址来识别,主机可以像访问主存一样去访问输入输出设备的寄存器。例如,VAX-11 机采用的就是统一编址方式,它把最高的 4KB 主存地址作为输入输出设备寄存器的地址。

6. 特权指令

特权指令是指具有特殊权限的指令,它用于多用户、多任务的计算机系统的系统资源分配与管理。这类指令的功能包括:改变系统的工作方式,检测用户的访问权限,修改虚拟存储器管理的段表、页表,完成任务的创建和切换等。为了安全,这类指令一般不直接提供给用户使用,如 Pentium 中的 SGDT、LSI、INVD 等。

7. 处理器控制指令

处理器控制指令是直接控制 CPU 实现某种功能的指令,包括状态标志位的操作指令、停机指令、等待指令、空操作指令、封锁总线指令等,如 Intel 8086 中的 STC、WAIT、NOP、LOCK。

3.2　指令格式

计算机的指令格式与机器的字长、存储器的容量及指令的功能都有很大关系。从便于程序设计、增加基本操作并行性、提高指令功能的角度来看,指令中应包含多种信息。但在有些指令中,由于部分信息可能无用,不仅浪费了指令所占的存储空间,增加了访存次数,可能还会影响处理速度。因此,如何合理、科学地设计指令格式,既给出足够的信息,又使指令长度尽可能地与机器字长相匹配,以节省存储空间,缩短取指时间,提高机器的性能,是指令格式设计中的一个重要问题。

计算机是通过执行指令来处理各种数据的。为了指出所执行的操作、数据的来源及操作结果的去向,一条指令必须包含下列信息。

(1) 操作码。它具体说明了操作的性质及功能。一台计算机可能有几十条至几百条指令,每条指令都有一个相应的操作码。计算机通过识别该操作码来完成不同的操作。

(2) 操作数的地址。CPU 通过该地址就可以取得所需的操作数。

(3) 操作结果的存储地址。把对操作数的处理结果保存在该地址中,以便再次使用。

(4) 下条指令的地址。执行程序时,大多数指令按顺序依次从主存中取出执行。只有在遇到转移指令时,程序的执行顺序才会改变。为了压缩指令的长度,可以用一个程序计数器存放指令地址。每取出一条指令,PC 中的指令地址就自动加 1(设该指令只占一个主存单元),指出将要取出的下一条指令的地址。当遇到转移指令时,则用转移地址修改 PC 的内容。由于使用了 PC,指令中就不必明显地给出下一条将要取出的指令的地址。

从上述分析可知，一条指令实际上包括两种信息，即操作码和地址码。因此，指令的结构可用如图 3-1 的形式来表示。

图 3-1 指令结构

3.2.1 指令字长

指令字长是指一条指令中所包含的二进制代码的位数，也就是指令的长度。

指令字长取决于操作码的长度、地址码的个数及长度。指令字长与机器字长没有固定的关系，它可以等于机器字长，也可以大于或小于机器字长。在字长较短的小型、微型机中，大多数指令的长度可能大于机器字长；而在字长较长的大、中型机中，大多数指令的长度则往往小于或等于机器字长。通常，把指令字长等于机器字长的指令称为单字长指令，指令字长等于半个机器字长的指令称为半字长指令，指令字长等于两个机器字长的指令称为双字长指令。例如 IBM 370 系列机，其指令格式有 16 位(半字)的，有 32 位(单字)的，还有 48 位(一个半字)的。

在指令系统中，若指令的长度随指令功能而异，称为变长指令结构。不同的指令可以有不同的长度，即需长则长，要短则短。但因为主存一般是按字节编址的，所以指令字长多为字节的整数倍。如在 Intel 80x86 中，指令长度是可变的，有 1 字节、2 字节、4 字节、8 字节，最长的有 12 字节。原则上讲，短指令比长指令好，主要是它能节省存储空间，提高取指令的速度，但有其局限性。长指令可能会占用两个或多个地址，取指令的时间相对来说就要长些，但可以扩大寻址范围或可以带几个操作数。两者各有所长，如果长、短指令可在同一机器中混合使用，就会给系统带来很大的灵活性。

到目前为止，主流 PC 和传统的大、中、小型机仍广泛采用复杂指令系统和变长指令格式。变长结构指令系统比较灵活，能充分利用指令字的每一位代码，但指令的控制较复杂。

在一个指令系统中，若所有指令的长度都是相等的，称为定长指令结构。定长结构的指令系统控制简单，但不够灵活。为了获得更高的执行速度，采取精简指令系统。它采用定长结构指令，可广泛用于工作站类的高档微机。也可采用众多的 RISC 处理器构成大规模并行处理阵列。

3.2.2 地址码

根据一条指令中地址码的数量，可将该指令称为几操作数指令或几地址指令。一般的操作数有源操作数、目的操作数及操作结果这三种数，因而就形成了三地址指令格式，这是早期计算机指令的基本格式。在三地址指令格式的基础上，后来又发展出二地址指令、一地址指令、零地址指令和多地址指令。各种不同操作数的指令格式如图 3-2 所示。

1. 三地址指令

三地址指令字中有三个地址码，其功能为

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

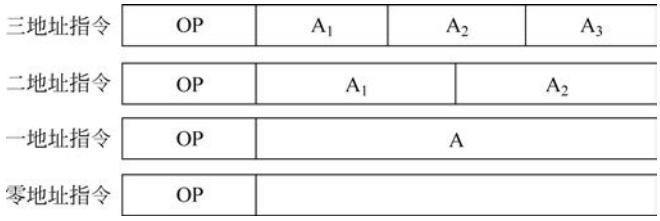

图 3-2　指令格式

式中，OP 表示操作性质，如加、减、乘、除等；A_1、A_2、A_3 可以是内存单元的地址，也可以是运算器中通用寄存器的地址，A_1 为源操作数地址，A_2 为目的操作数地址，A_3 为存放操作结果的地址；←表示把操作结果传送到指定的地方。

这种格式的指令长度比较长，小型、微型机中很少使用。

2. 二地址指令

二地址指令常称为双操作数指令，它的两个地址码分别指明参与操作的两个操作数存放的内存单元地址或通用寄存器的名称。A_1 为源操作数地址，A_2 为目的操作数地址，A_2 同时兼作存放操作结果的地址。指令执行之后，A_2 地址中原来存放的内容已被覆盖了，其功能为

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

在二地址指令格式中，从操作数的物理位置来说，又可归结为三种类型。

(1) 存储器-存储器(SS)型指令：操作数都存放在内存单元中。执行指令时，需从内存某单元中取出操作数，并将操作结果存放至内存另一单元中，因此机器执行这种指令需要多次访问内存。

(2) 寄存器-寄存器(RR)型指令：可使用通用寄存器组或个别专用寄存器存放操作数。执行指令时需从寄存器中取出操作数，并把操作结果放到另一寄存器中。机器执行寄存器-寄存器型指令的速度很快，因为执行这类指令不需要访问内存。

(3) 寄存器-存储器(RS)型指令：一个操作数存放在通用寄存器组的某个寄存器中，另一个操作数存放在内存单元中。执行此类指令时，既要访问内存单元，又要访问寄存器。

二地址指令是最常见的指令格式，在各种计算机的指令系统中广泛采用。

3. 一地址指令

一地址指令中只给出一个地址，该地址既是操作数的地址，又是操作结果的存储地址，如加 1、减 1 和移位等单操作数指令均采用这种格式。其操作过程是：对这一地址所指定的操作数执行相应的操作后，得出的结果又存回该地址中，其功能为

$$A_1 \leftarrow OP(A_1)$$

另一种情况是以 CPU 中的累加寄存器(AC)的内容作为一个隐含操作数，指令的地址码 A 指明的是另一个操作数，操作结果又送回 AC，其功能为

$$AC \leftarrow (AC)\,OP(A)$$

4. 零地址指令

零地址指令格式只有操作码字段。这种格式可用于只需操作码字段而不需要操作数的指令，如停机、空操作、清除等控制指令。

但某些需要操作数的算术、逻辑类指令也可以用零地址指令来实现，这时操作数地址是隐含的。如对 AC 内容进行操作，其 AC 为隐含约定时可用零地址。堆栈计算机中的算术、逻辑类指令就是零地址指令，此时参加运算的操作数放在堆栈中，运算结果也放在堆栈中（有关堆栈的详细内容在后面介绍）。

5. 多地址指令

某些性能较好的大、中型机甚至高档小型机中，往往设置了一些功能很强的、用于处理成批数据的指令，如字符串处理指令、向量和短阵运算指令等。为了描述一批数据，指令中需要多个地址来指出数据存放的首地址、长度和下标等信息。例如 CDC STAR-100 计算机系统的矩阵运算指令，其地址码部分有 7 个地址段，用来指出两个矩阵的数据存放情况及结果的存放情况。

指令中地址个数的选取要考虑诸多因素。从缩短程序长度、方便用户使用和增加操作并行度等方面来看，选用多地址指令格式较好；从缩短指令长度、减少访存次数、简化硬件设计等方面来看，一地址指令格式较好。对于同一个问题，用三地址指令编写的程序行数最少，但程序目标代码的长度最长；而用二、一、零地址指令来编写程序，程序行数一个比一个多，但程序目标代码的长度一个比一个短。表 3-1 给出了不同地址数指令的特点及适用场合。

表 3-1 不同地址数指令的特点及适用场合

地址数量	程序长度	程序存储空间	执行速度	适 用 场 合
三地址	短	最大	一般	向量、矩阵运算为主
二地址	一般	大	较慢	广泛使用
一地址	较长	较大	较快	连续运算，硬件结构简单
零地址	较长	最小	最快	嵌套、递归问题

3.2.3 操作码

指令系统的每一条指令都有一个唯一的、用二进制代码表示的操作码，它用来表示该指令应进行什么性质的操作，如进行加法、减法、乘法、除法、数据传送等，其长度取决于指令系统中的指令条数和编码格式。

指令操作码主要有两种编码格式：固定长度格式和可变长度格式。

1. 固定长度操作码

固定长度操作码指操作码的长度固定，且集中放在指令字的一个字段中。这种格式对于简化硬件设计、减少指令译码时间非常有利，在字长较长的大、中型机和超级小型机以及 RISC 上被广泛采用。若某机器的操作码字段长度为 K 位，则它最多能表示 2^K 条不同指

令。如 IBM 370 机(字长 32 位)的操作码字段都是 8 位,8 位操作码允许指定 256 条指令。而实际上在 IBM 370 机中只有 183 条指令,存在着较大的信息冗余。

IBM 370 机的指令可分为三种不同的长度形式:半字长指令、单字长指令和一个半字长指令。半字长指令不包含主存地址;单字长指令只指明一个主存地址;一个半字长指令则给出两个主存地址,共有 5 种格式,如图 3-3 所示。

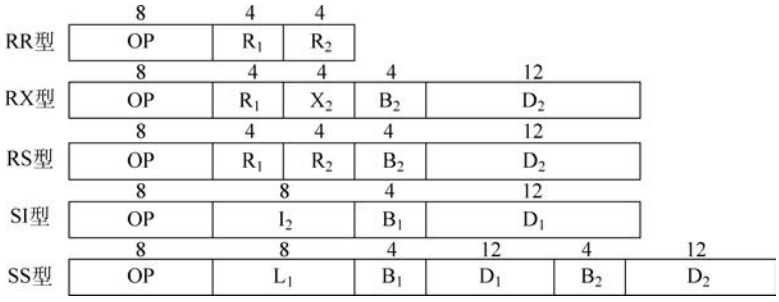

图 3-3　IBM 370 机的指令格式

【例 3.1】 机器的指令字长为 16 位,指令格式如图 3-4 所示,其中 OP 为操作码,试分析该指令格式的特点。

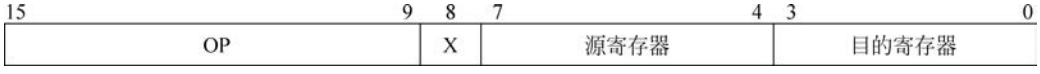

图 3-4　例 3.1 图

解　由图 3-4 可知:

(1) 该指令为单字长二地址指令;

(2) 操作码字段 OP 为 7 位,最多能表示 128 条不同指令;X 字段用于寻址方式;

(3) 源寄存器和目标寄存器都是通用寄存器(可分别指定 16 个),所以是 RR 型指令,两个操作数均存放在寄存器中;

(4) 这种指令结构常用于算术、逻辑运算类指令。

【例 3.2】 机器的指令字长为 16 位,指令格式如图 3-5 所示,OP 为操作码字段,试分析指令格式的特点。

图 3-5　例 3.2 图

解　由图 3-5 所知:

(1) 该指令为双字长二地址指令,用于访问存储器;

(2) 操作码字段 OP 为 7 位,最多能表示 128 条不同指令;X 字段用于寻址方式;

(3) 一个操作数存放在源寄存器(共 16 个),另一个操作数存放在存储器中(由变址寄存器和位移量决定),所以是 RS 型指令。

2. 可变长度操作码

可变长度操作码不但长度可变,而且分散地存放在指令字的不同字段中。具体方法是:

当指令中的地址码位数较多时，让操作码的位数少些；当指令中的地址码位数减少时，让操作码的位数增多，以增加指令种类，称为扩展操作码。这样既能充分利用指令字的各个字段，又能在不增加指令长度的情况下扩展操作码的长度，使它能表示更多的指令。这种格式在字长较短的小型和微型机上广泛采用，如 PDP-11/VAX-11、Intel 80x86 和 Pentium 等。PDP-11 机的指令分为单字长、二字长、三字长三种，操作码字段长度为 4～16 位不等，可遍及整个指令长度。PDP-11 机的部分指令格式如图 3-6 所示。

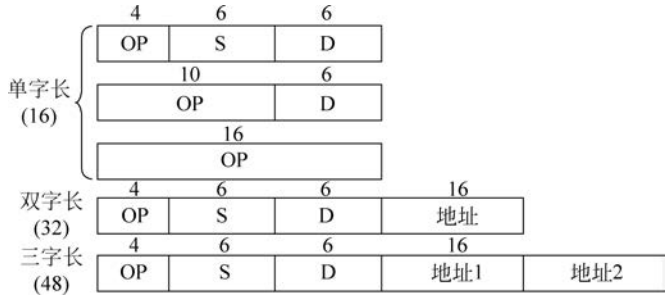

图 3-6 PDP-11 机的部分指令格式

显然，操作码长度不固定将增加指令译码和分析的难度，使控制器的设计复杂化，因此对操作码的编码至关重要。通常是在指令字中用一个固定长度的字段来表示基本操作码。

例如，设某机器的指令字长为 16 位，包括 1 个 4 位的基本操作码字段和 3 个 4 位地址码字段，其格式如图 3-7 所示。

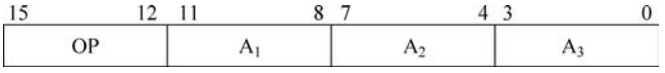

图 3-7 某机器的指令格式

4 位基本操作码有 16 种编码组合，若全部用三地址指令表示，则有 16 条。但是，若三地址指令仅需 15 条，两地址指令需 15 条，一地址指令需 15 条，零地址指令需 16 条，共 61 条指令，应如何安排操作码？显然，只有 4 位基本操作码是不够的，必须将操作码的长度向地址码字段扩展才行。一种可供扩展的方法和步骤如下：

(1) 15 条三地址指令的操作码用 4 位基本操作码(0000～1110)表示，剩下的 1111 用于把操作码扩展到 A_1，即从 4 位扩展到 8 位；

(2) 15 条二地址指令的操作码用 8 位操作码(11110000～11111110)表示，剩下的 11111111 用于把操作码扩展到 A_2，即从 8 位扩展到 12 位；

(3) 15 条一地址指令的操作码用 12 位操作码(111111110000～111111111110)表示，剩下的 111111111111 用于把操作码扩展到 A_3，即从 12 位扩展到 16 位；

(4) 16 条零地址指令的操作码用 16 位操作码(1111111111110000～1111111111111111)表示。

一般将指令操作码放在指令字的第一字节，当读出操作码后就可马上判定这是一条单操作数指令还是一条双操作数指令，或者是零地址指令，从而知道后面还应该取几个字节指令代码。当然，在采取预取指令技术时，这个问题会复杂一些。

实际上，在可变长度操作码的指令系统设计中有一个重要的原则，就是使用频率高的指令应分配短的操作码，使用频率低的指令相应地分配较长的操作码。哈夫曼

(Huffman)编码法就是根据上述原则进行编码的。这样不仅可以有效地缩短操作码在程序中的平均长度,节省存储器空间,而且缩短了使用频率高的指令的译码时间,提高了程序的运行速度。

【例 3.3】 设指令字长为 12 位,每个地址码为 3 位,采用扩展操作码的方式,设计 4 条三地址指令、255 条一地址指令和 8 条零地址指令。

要求:

(1) 写出扩展表示;

(2) 画出指令译码逻辑图;

(3) 计算操作码的平均长度。

解 (1) 操作码的扩展表示如下:

```
0 0 0 × × × × × × × × ×  ┐
        ⋮                 ⋮ ├ 4 条三地址指令
0 1 1 × × × × × × × × ×  ┘
1 0 0   0 0 0   0 0 0 × × ×  ┐
          ⋮       ⋮          ├ 255 条一地址指令
1 1 1   1 1 1   1 1 0 × × ×  ┘
1 1 1   1 1 1   1 1 1 0 0 0  ┐
          ⋮       ⋮          ├ 8 条零地址指令
1 1 1   1 1 1   1 1 1 1 1 1  ┘
```

(2) 指令译码逻辑如图 3-8 所示。

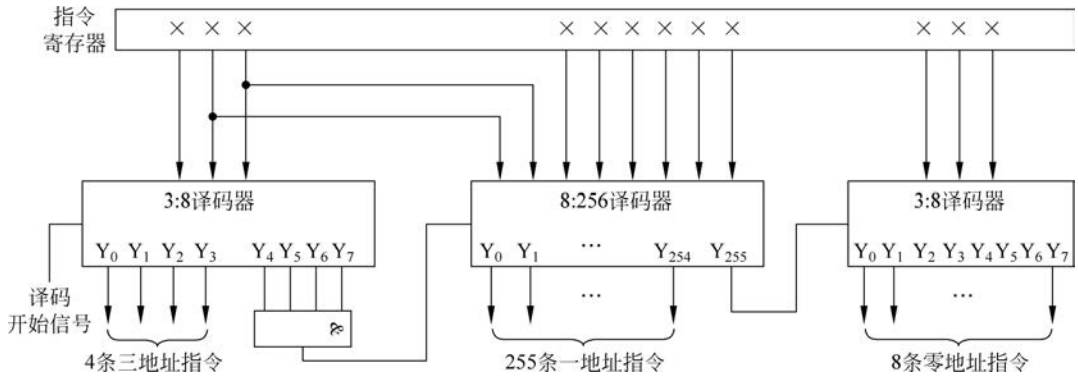

图 3-8　指令译码逻辑图

(3) 指令操作码的平均长度=(4×3+9×255+12×8)/267=9(位)。

3. 指令助记符

由于计算机只能识别 1 和 0,所以采用二进制操作码是必要的,但是用二进制代码来书写程序却非常麻烦。为了便于书写和阅读程序,每条指令通常用 3 个或 4 个英文缩写字母来表示,这种缩写码叫作指令助记符,如表 3-2 所示。这里假定指令系统只有 7 条指令,所以操作码只需 3 位二进制。

表 3-2 典型的指令助记符

典 型 指 令	指令助记符	二进制操作码
加法	ADD	001
减法	SUB	010
数据传送	MOV	011
跳转	JMP	100
转子程序	CALL	101
存操作数	STR	110
取操作数	LDA	111

由于指令助记符提示了每条指令的意义，因此比较容易记忆，书写起来比较方便，阅读时也容易理解。例如一条加法指令可以用助记符 ADD 来代表操作码 001；而一条数据传送指令可以用助记符 MOV 来代表操作码 011。

由指令助记符构成的指令称为汇编指令。在讨论指令系统时为了方便，经常使用汇编指令。需要注意的是，在不同的计算机中，指令助记符的规定是不一样的。如 80x86 系列 CPU 的汇编指令与 MCS-51 单片机的汇编指令是不同的。

计算机只能识别和执行二进制语言，因此，汇编指令还必须转换成与它们相对应的二进制码。这种转换借助汇编程序可以自动完成，汇编程序相当于一个“翻译”。

3.3 数据的存储与寻址方式

存储器中存储的数据包括两大类：一是指令，二是操作数。对这些数据的存储和寻址，不同的计算机会有不同的方法。

3.3.1 数据的存储方式

1. 操作数的存储方式

机器指令可以对不同类型的操作数进行操作。操作数的类型有数值数据和非数值数据，数值数据可以是定点整数、浮点数、十进制数，操作数的位数可以是字节、字、双字、四字。

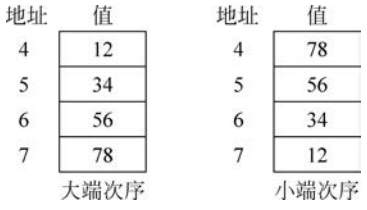

图 3-9 多字节数据的存储方式

一个多字节的数据在按字节编址的主存中通常有两种排序方式：大端次序(big-endian ordering)和小端次序(little-endian ordering)。大端次序将最高有效字节存储在最小地址单元，小端次序将最低有效字节存储在最小地址单元。图 3-9 是 4 字节的十六进制数 12345678 在存储器中的存储方式示意图。

采用大端次序的优点如下。

(1) 字符串排序方便。因为字符串与整型数据的字节顺序一致，在对大端次序字符串进行比较时，整型 ALU 可以同时比较多个字符。

(2) 顺序一致。采用相同的顺序存储整型数和字符串时，显示十进制数以及字符串很方便，所有的数值可直接从左到右顺序显示而不会产生混淆。

小端次序的优点是其适合于超长数据的算术运算。对采用小端次序存储的数据进行高精度算术运算时，不需要寻找最低字数据，后再反向移动到较高位。

Intel 80x86、Pentium、VAX 和 Alpah 是采用小端次序的处理器，IBM S370/390、Motorola 的 680x0、Sun SPARC 和大多数 RISC 机器则采用大端次序，Power PC 是既支持大端次序又支持小端次序的机器。当数据由一种端序类型的机器传送到另一种类型的机器，或当程序试图对多字节数据的个别字节或个别位进行处理时，要特别注意数据的存储次序。

2. 数据对齐方式

在数据对齐存储方式下，要求一个数据字占据完整的一个字的存储位置，而不是分成两部分各占据一个字存储位置的一部分。例如一个 32 位的数据字，在按字对齐方式下，它的地址应当是 4 的倍数，即其地址的二进制码的最低两位为 00。这样，它实际占据的存储单元的地址为 $4n$、$4n+1$、$4n+2$ 和 $4n+3$（n 为自然数）。在 32 位宽度的存储器中，因为其字地址为 $4n$，这个数据可以一次读取或者写入。如果这个数据字不按对齐方式存储，其存储单元的地址会出现如 $4n-1$、$4n$、$4n+1$ 和 $4n+2$ 的情况，这样的数据在 32 位的存储器需要分两次读取或者写入。在计算机中，规定数据的存储必须按字对齐的方式进行。图 3-10(a)是一个字未对齐的例子，图 3-10(b)是一个字对齐的例子。所以在存储器中有些位置是空白位置，这种空白存储位置是为了保证后继数据按字对齐方式进行存储。

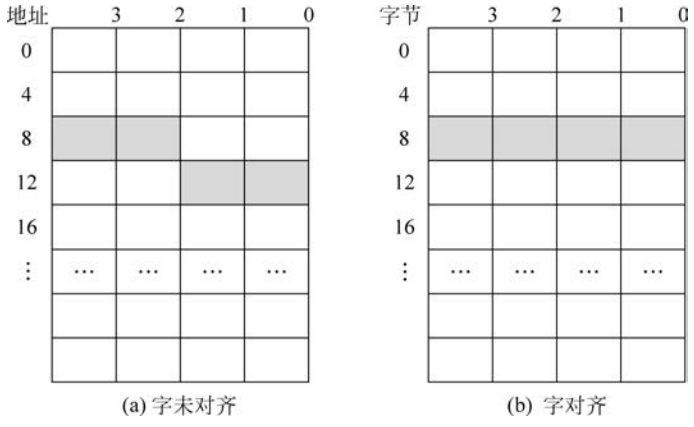

图 3-10　存储器中数据字的对齐方式

3.3.2　寻址方式

计算机中有两种信息，即指令和数据（或称操作数），它们都存放在存储器相应的存储单元中。运行程序时，计算机会逐条执行指令，并对数据进行处理。那么，如何从存储器中找到所需要的指令或数据呢？很明显，只要找到它们在存储器中的有效地址即可。所谓寻址方式，就是形成指令或数据有效地址的方法。

寻址方式是指令系统设计的重要内容。一套好的寻址方式能给用户提供丰富的程序设计手段，能提高程序的质量和存储空间的利用率。寻址方式是不同计算机的重要特色之一，它们对寻址方式的分类及寻址方式的名称有各自的规定。下面介绍大多数计算机都使用的

基本寻址方式，其中有些基本寻址方式可以组合使用，从而形成更复杂的寻址方式。

1. 指令寻址方式

指令寻址方式分为顺序寻址方式和跳跃寻址方式两种。

(1) 顺序寻址方式。组成程序的指令序列在主存中是顺序存放的。因此，执行程序是从该程序的第一条指令开始逐条取出并逐条执行。这种程序的顺序执行过程称为顺序寻址方式。为了达到顺序寻址的目的，CPU 中可以用一个程序计数器(program counter，PC)对指令的地址进行顺序计数。

PC 中开始时存放程序的首地址，然后每取出一条指令，PC 就加 1 或加一个常数，以指出下一条指令的地址，直到程序结束。指令顺序寻址方式的过程如图 3-11 所示。

(2) 跳跃寻址方式。当程序中出现分支或循环时，就会改变程序的执行顺序。此时，对指令寻址就要采取跳跃寻址方式。所谓跳跃，就是指下一条指令的地址不是通过 PC 加 1 获得的，而是由指令本身给出的。指令系统中的无条件转移指令和各种条件转移指令就是为跳跃寻址方式设置的。跳跃寻址方式的过程如图 3-12 所示。

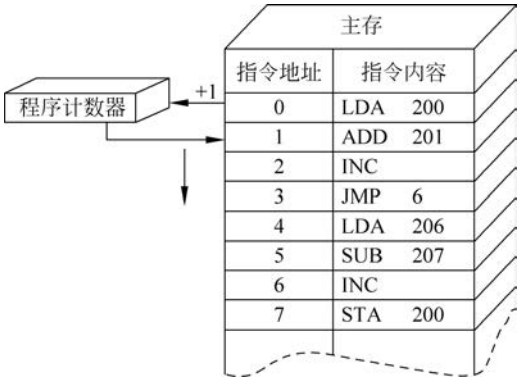

图 3-11 指令顺序寻址方式

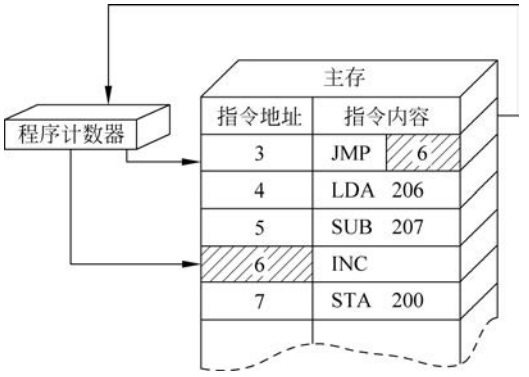

图 3-12 指令跳跃寻址方式

当执行 JMP 指令时，PC 不加 1，而是将 JMP 指令中的地址码 06 送到 PC；计算机从 06 地址取出指令执行，之后又顺序执行，直到再遇到转移指令或程序结束。

2. 操作数寻址方式

操作数寻址方式就是形成操作数有效地址的方法。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，因此把指令中用地址码字段给出的地址称为形式地址。这个地址有可能无法直接用来访问主存，而经过某种运算后可得到能够直接访问主存的地址，称为有效地址(effective address，EA)。

设某计算机指令系统的单地址指令结构如图 3-13 所示。其中，OP 为操作码，X 为寻址特征码，D 为形式地址(或称偏移量)。寻址过程就是把 X 和 D 的不同组合变换成有效地址的过程。常用的寻址方式有立即寻址、直接寻址、间接寻址、相对寻址、寄存器寻址、变址寻址等。

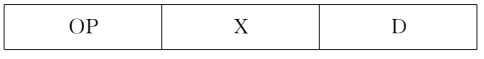

图 3-13 单地址指令结构

(1) 立即寻址。指令的地址码字段指出的不是操作数的地址，而是操作数本身，这种寻址方式称为立即寻址(immediate addressing)。这种寻址方式的特点是：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，从而提高了指令的执行速度。但是，因为操作数是指令的一部分，不能被修改，而且立即数(立即寻址方式指令中给出的数)的大小受到指令长度的限制，所以这种寻址方式灵活性较差，通常用于给某一寄存器或主存单元赋初值。例如 Intel 80x86 的 MOV AX，2000H 指令，表示把立即数 2000H 传送给累加寄存器 AX。

(2) 直接寻址。指令的地址码字段直接指出操作数的有效地址，这种寻址方式称为直接寻址(direct addressing)。此时，特征码 X 指出寻址方式是直接寻址方式，形式地址 D 给出操作数的地址。若用 EA 代表有效地址，则 EA＝D。例如 Intel 80x86 的 MOV AX，[2000H]指令，表示源操作数的有效地址 EA＝2000H。直接寻址方式的过程如图 3-14 所示。

这种寻址方式不需要作任何寻址运算，简单直观，也便于硬件实现。但随着计算机主存容量的不断扩大，所需的地址码将会越来越长。对于定长指令，用于表示地址码字段的位数有限，限制了访问主存的范围；而对于变长指令，势必造成指令的长度太长。因此这种寻址方式缺少灵活性。

(3) 间接寻址。间接寻址(indirect addressing)是相对直接寻址而言的。在间接寻址方式中，地址字段中的形式地址不是操作数的有效地址，而是操作数地址的地址。就是说，D 指示的存储单元中的内容才是操作数的有效地址，而 D 只是一个间接地址。此时，特征码 X 指出寻址方式为间接寻址方式，有效地址 EA＝(D)。间接寻址方式的过程如图 3-15 所示。

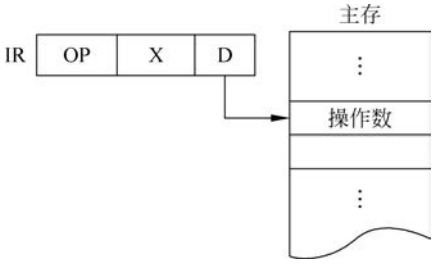

图 3-14　直接寻址方式

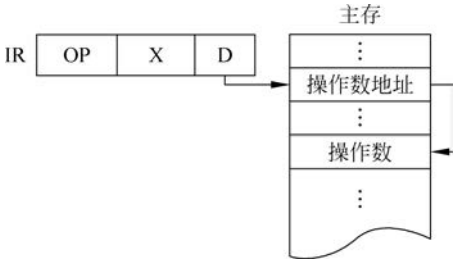

图 3-15　间接寻址方式

间接寻址扩大了寻址范围，可用较短的地址访问较大的主存空间。例如某计算机指令字长为 16 位，指令中的地址码为 10 位，直接寻址空间为 2^{10}(＝1K) 个存储单元。采用一级间接寻址后，从存储器中将取出 16 位的有效地址，能够访问 2^{16}(＝64K)个存储单元。

间接寻址中又有一级间址和多级间址之分，图 3-15 所示的间接寻址是一级间址。采用多级间址时，为取得操作数需要多次访问主存。对于多级间址来说，在寻址过程中所访问的每个主存单元的内容中都应设置一个间址标志位。通常将这个标志位放在主存单元的最高位。当该位为“1”时，表示这一主存单元中仍然是间接地址，需要继续间接寻址；当该位为“0”时，表示已经找到了有效地址，根据这个地址可以找到真正的操作数。间接寻址在取指之后至少需要两次访问主存才能取出操作数，降低了指令执行的速度。所以，大多数计算机只允许一级间址。

(4) 寄存器寻址。寄存器寻址(register addressing)是在指令的地址码字段中给出某一个通用寄存器的编号,这个寄存器中存放着操作数。例如 Intel 80x86 的 MOV AX,DX 指令,表示源操作数存放在 DX 寄存器中,把它取出后就可以传送给累加寄存器 AX。寄存器寻址方式的过程如图 3-16 所示。

寄存器寻址的优点是:从寄存器中存取数据比从主存中快得多;由于寄存器的数量较少,其地址码字段较短,因此缩短了指令长度,提高了指令的执行速度,在各种计算机中广泛使用。

(5) 寄存器间接寻址。寄存器间接寻址(register indirect addressing)是在指令的地址码中给出某一通用寄存器的编号,在寄存器中存放操作数的有效地址,而操作数则存放在主存单元中。此时,X 特征码表示寻址方式为寄存器间接寻址方式,有效地址 EA=(R)。例如 Intel 80x86 的 MOV AX,[BX]指令,表示源操作数的有效地址存放在 BX 寄存器中。寄存器间接寻址方式的过程如图 3-17 所示。

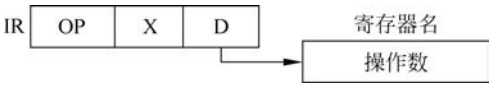

图 3-16 寄存器寻址方式

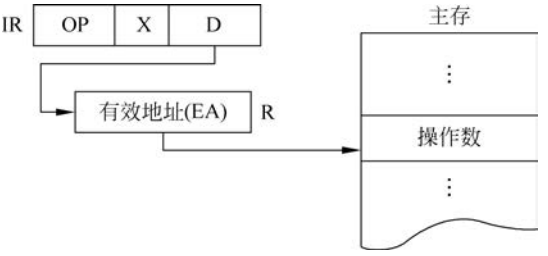

图 3-17 寄存器间接寻址方式

寄存器间接寻址方式的指令代码较短,克服了间接寻址中访存次数多的缺点,指令执行速度快,是一种广泛使用的寻址方式。此时寄存器本身被称为间接地址指示器,在编程过程中常作为地址指针。

(6) 相对寻址。相对寻址(displacement addressing)是把 PC 中的内容加上指令中的形式地址 D,形成操作数的有效地址。此时,特征码 X 表示寻址方式为相对寻址方式,PC 中的内容是当前的指令地址。这里的“相对”,就是相对当前的指令地址,所以,操作数的有效地址为 EA=(PC)+D。形式地址 D 可正、可负,通常用补码表示;操作数可在指令存储单元之前或之后。操作数的地址与指令地址之间总是相差一个固定值,支持程序在主存中任意浮动。相对寻址方式的过程如图 3-18 所示。

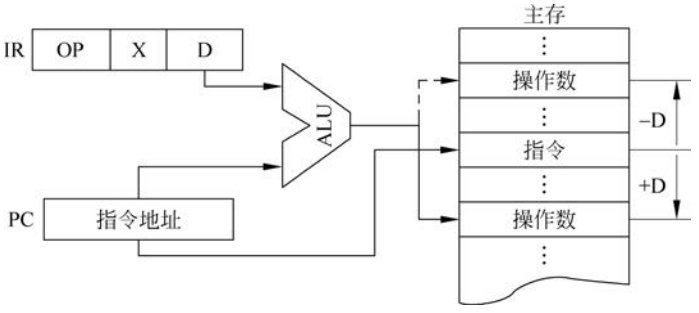

图 3-18 相对寻址方式

(7) 变址寻址。将变址寄存器的内容加上位移量 D 形成操作数的有效地址 EA,称为变址寻址(index addressing)。这种寻址方式与相对寻址的区别在于它与本条指令的地址

无关，操作数地址随变址寄存器的内容浮动一个 D。变址寄存器的内容可由程序员设置，故寻址更灵活。在具有变址寻址的指令中，除去操作码和形式地址外，还必须指明变址寄存器。变址寻址方式的过程如图 3-19 所示。

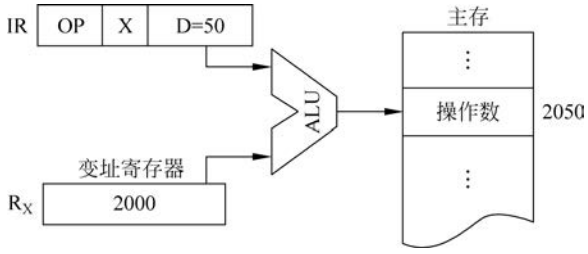

图 3-19　变址寻址方式

变址寻址是一种广泛采用的寻址方式，最典型的用法是将指令中的形式地址作为基准地址，而变址寄存器的内容作为修改量。当需要频繁修改地址时无须修改指令，只要修改变址值就可以了，这对于数组运算、字符串操作等成批数据的处理是很有用的。例如 Intel 80x86 的 MOV AX，TABLE[SI]指令，表示源操作数的有效地址 EA＝(SI)＋TABLE，其中 SI 是变址寄存器，TABLE 是数组或字符串的基准地址，在此作为位移量 D。

(8) 基址寻址。基址寻址(base addressing)是将基址寄存器 R_b 的内容与指令中给出的位移量 D 相加，形成操作数的有效地址，即 EA＝(R_b)＋D。当特征码 X 指明是基址寻址时，要指定一个寄存器来存放寻址的基准值，这个寄存器称为基址寄存器。指令的地址码字段是一个可正、可负的位移量。基址寻址方式的过程如图 3-20 所示。

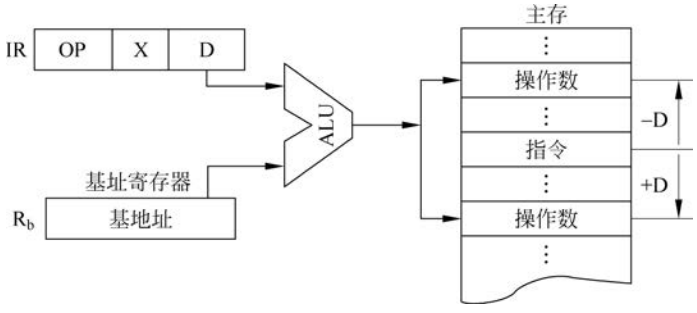

图 3-20　基址寻址方式

基址寻址和变址寻址在形成有效地址时所用的方法是相同的，而且在一些计算机中，这两种寻址方式都是用同样的硬件来实现的。两者的区别是应用的场合不同，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；而基址寻址是面向系统的，主要用于逻辑地址和物理地址间的变换，用以解决程序在主存中的再定位和扩大寻址空间等问题。

(9) 段寻址。段寻址(segment addressing)是一种为了扩大寻址范围而采用的技术，在 Intel 80x88/80x86 等处理器中使用。80x88/80x86 处理器字长为 16 位，因此寻址范围只有 2^{16}＝64KB，而处理器的直接寻址范围可达到 2^{20}＝1MB，其中的奥妙就是采用了段寻址方式。具体方法是将 1MB 的存储空间以 64KB 为单位分为若干段，在形成操作数的有效地址时，由一个基地址加上某寄存器提供的 16 位偏移量形成 20 位物理地址，这个基地址由

CPU 中的段寄存器提供；形成 20 位物理地址后，段寄存器中的内容自动左移 4 位，然后与偏移量相加，即形成所需的 20 位地址。段寻址方式的过程如图 3-21 所示。

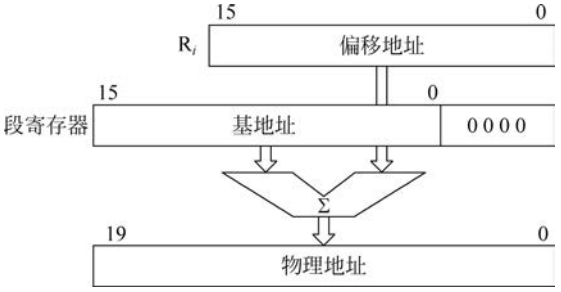

图 3-21 段寻址方式

(10) 复合寻址。复合寻址(composite addressing)是把两种或两种以上的基本寻址方式组合在一起形成的寻址方式。如间接寻址方式与变址寻址或相对寻址方式结合形成的寻址方式。

以上介绍了一些常用的基本寻址方式。对于某类型计算机的指令系统，可能只用了上述寻址方式的一部分或某些基本方式再加上几种变形的寻址方式。只要熟悉了以上的寻址方式，其他的寻址方式是很容易理解的。另外，在双地址指令或多地址指令中，各地址码的寻址方式可能不同，请读者注意。

【例 3.4】 一种二地址 RS 型指令的结构如图 3-22 所示。

6位		4位	1位	2位	16位
OP	—	通用寄存器	I	X	位移量(D)

图 3-22 二地址 RS 型指令的结构

其中 I 为间接寻址标志位，X 为寻址模式字段，D 为位移量字段。通过 I、X、D 的组合，可构成如表 3-3 所示的寻址方式。请写出表 3-3 中 6 种寻址方式的名称。

表 3-3 寻址方式

寻址方式	I	X	有效地址 EA 的算法	说 明
(1)	0	00	EA=D	
(2)	0	01	EA=(PC)+/−D	PC 为程序计数器
(3)	0	10	EA=(R_2)+/−D	R_2 为变址寄存器
(4)	1	11	EA=(R_3)	
(5)	1	00	EA=(D)	
(6)	0	11	EA=(R_1)+/−D	R_1 为基址寄存器

解 (1) 直接寻址；(2) 相对寻址；(3) 变址寻址；
(4) 寄存器间接寻址；(5) 间接寻址；(6) 基址寻址。

3. 堆栈寻址方式

最后介绍堆栈寻址方式。堆栈(stack)是一种数据项按序排列的线性序列，它的存取顺序是“后进先出”(last in first out)。堆栈是一个预留的位置块，数据项是被陆续加到栈顶

的。在任何时刻，堆栈的位置块都是部分被填充的。堆栈寻址方式是一种隐含寻址，隐含地指示对栈顶位置的数据执行操作。

1）堆栈的设置

堆栈的设置通常有两种方式：硬堆栈和软堆栈。硬堆栈由 CPU 中的一组专门寄存器组成，容量有限。软堆栈通过执行相关指令，把主存储器中的一段存储区定义为堆栈，利用一个通用寄存器作为堆栈指针 SP(stack pointer)，并设置 SP 的初值。目前大多数计算机都支持软堆栈，软堆栈设置灵活，可满足较大容量的要求，可以用对存储器寻址的指令来对堆栈中的数据进行访问。

2）堆栈的工作方式

将数据存入堆栈的操作称为“进栈(PUSH)”，从堆栈中取出数据的操作称为“出栈(POP)”，出栈后相应单元的数据不再保留。最先进栈的数据位于堆栈的底端(栈底)，最后进栈的数据位于堆栈的顶端(栈顶)，进栈和出栈操作均在栈顶进行。图 3-23 展示了“进栈”操作的过程。

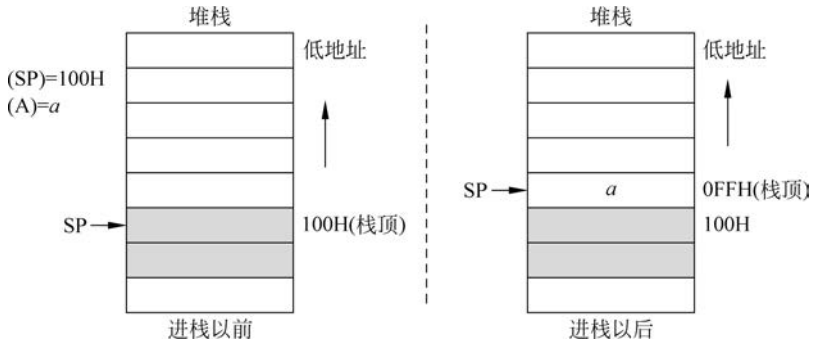

图 3-23　进栈操作

在执行进栈操作前，设堆栈指针(SP)＝100H，通用寄存器(A)＝a。假如现在把通用寄存器 A 中的数据送入堆栈，那么执行“进栈”指令后，SP 会自动“减 1”成为 0FFH(十六进制)；数据 a 被压入存储单元 0FFH，SP 始终指向栈顶(最后存入数据的堆栈单元)。因此进栈操作可描述为

$$SP \leftarrow (SP) - 1, M_{SP} \leftarrow (A)$$

其中(A)表示通用寄存器 A 的内容；SP 表示堆栈指针；M_{SP} 表示堆栈指针指示的栈顶单元。

图 3-24 展示了“出栈”操作的过程。

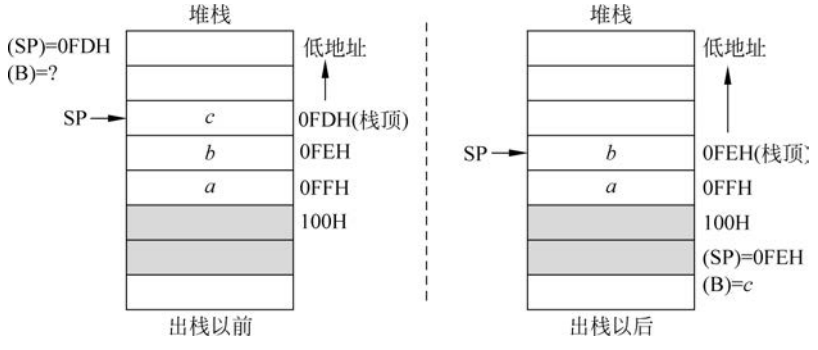

图 3-24　出栈操作

假定出栈操作之前，堆栈中已存入 a、b、c 三个数，堆栈指针(SP)＝0FDH。要求把堆栈中的数据送到通用寄存器 B，那么执行“出栈”操作后，就把当前栈顶单元的内容 c 送到了 B，堆栈指针 SP 变为 0FEH。

出栈操作可描述为

$$B \leftarrow (M_{SP}), SP \leftarrow (SP) + 1$$

要注意的是，有的机器执行进栈操作时，SP 由低地址向高地址变化；出栈操作时，SP 由高地址向低地址变化。

除了利用堆栈暂时存取数据外，在执行转子程序指令、中断指令时也要用到堆栈，子程序的嵌套、递归调用也离不开堆栈。

3.4 指令集架构

本节介绍 CISC 和 RISC 的发展及特点，阐述指令集架构的定义，讨论 Intel 80x86 和 RISC-V 等典型指令集架构的变化发展。

3.4.1 CISC 和 RISC

从计算机指令系统设计的角度看，计算机的体系结构可分为 CISC 和 RISC。CISC 对计算机体系结构的发展功不可没，而 RISC 则是近 30 年来迅速发展起来的一种体系结构。

1. CISC

随着集成电路技术的发展，计算机的硬件成本不断下降，但软件成本不断提高。为此，计算机系统设计者在设计指令系统时，增加了越来越多的功能强大的复杂指令以及更多的寻址方式，以满足不同方面的需求。

(1) 更好地支持高级语言。增加语义接近高级语言语句的指令，缩短指令系统与高级语言之间的语义差距。

(2) 简化编译。编译器是将高级语言翻译成机器语言的软件。当机器指令的语义与高级语言的语义接近时，编译器的设计变得相对简单，编译的效率也会大大提高，而且编译后的目标程序也能得到优化。

(3) 满足系列计算机软件向后兼容的需求。为了做到程序兼容，同一系列计算机的新型号计算机和高档计算机的指令系统只能扩充而不能减少原来的指令，因此指令数量越来越多。

(4) 更好地支持操作系统。随着操作系统的功能越来越复杂，要求指令系统提供对相应功能指令的支持，如多媒体指令和 3D 指令等。

(5) 压缩地址码长度，设计多种寻址方式。为在有限指令长度内基于扩展法实现更多指令，只有最大限度地压缩地址码长度。但为满足寻址访问的需要，必须设计多种寻址方式，如基址寻址、相对寻址等。

基于上述原因，指令系统越来越庞大，越来越复杂，某些计算机中的指令多达数百条，同时寻址方式的种类也很多，称这类计算机为 CISC。Intel 80x86、IA64 指令系统就是典型的

CISC 指令系统。

CISC 具有如下特点：

(1) 指令系统复杂庞大，指令数目一般多达二三百条；

(2) 寻址方式多；

(3) 指令格式多；

(4) 指令字长不固定；

(5) 对访存指令不加限制；

(6) 各种指令使用频率相差大；

(7) 各种指令执行时间相差大；

(8) 大多数采用微程序控制器。

2. RISC

1979 年，以 Patterson 教授为首的一批计算机科学家对 CISC 进行了详细的研究，得到了以下结果。

首先，在 CISC 计算机中，各种指令的使用频率相差悬殊：一个典型程序在执行过程中所使用的 80%的指令类型，只占一个处理器指令系统所有类型的 20%。事实上最频繁使用的指令是存数、取数和算术运算这些最简单的指令。这样一来，长期致力于复杂指令系统的设计，实际上是在设计一种难得在实践中用得上的指令系统的处理器。同时，复杂的指令系统必然带来结构的复杂性，这不但增加了设计的时间与成本，还容易造成设计失误。

此外，许多复杂指令需要极复杂的操作，这类指令多数是某种高级语言的直接翻版，因而通用性差。由于采用二级的微程序执行方式，它也降低了那些被频繁使用的简单指令的运行速度。

因此，针对 CISC 的这些弊病，美国加州大学伯克利分校由 Patterson 教授领导的研究小组，首先提出了 RISC 这一术语，即指令系统应当只包含那些使用频率很高的少量指令，并提供一些必要的指令以支持操作系统和高级语言。按照这个原则发展而成的计算机被称为精简指令集计算机，简称 RISC。Patterson 教授领导的研究小组先后研制了 RISC-Ⅰ和 RISC-Ⅱ计算机。1981 年，美国斯坦福大学在 Hennessy 教授领导下的研究小组研制了 MIPS RISC 计算机，强调高效的流水线和采用编译方法进行流水调度，使 RISC 技术设计风格得到很大补充和发展。

RISC 具有以下特点。

(1) 充分利用 VLSI 芯片的面积。CISC 机器的控制器大多采用微程序控制，其控制存储器在 CPU 芯片内所占的面积为 50%以上；而 RISC 机器的控制器采用组合逻辑控制，其硬布线逻辑只占 CPU 芯片面积的 10%左右，可将空出的面积供其他功能部件使用。这样，在相同的集成电路制造工艺条件下，同样的芯片面积上可实现更多的并行部件，如超级流水线、多核处理器。

(2) 提高计算机的运算速度。RISC 对运算速度的提高主要反映在以下 5 个方面：①RISC 指令系统采用固定长度的指令格式，选取使用频率最高的一些简单指令，指令种类少，指令规整、简单，基本寻址方式只有 2～3 种。②RISC 机器内大量使用寄存器，数据处理指令只对寄存器进行操作，只有加载/存储指令可以访问存储器，可提高指令的执行效率。

③RISC 机器采用寄存器窗口重叠技术，因此，程序嵌套时不需要将寄存器内容保存到存储器中，提高了执行速度。④RISC 机器采用组合逻辑控制，比采用微程序控制的 CISC 机器的延迟小，缩短了 CPU 周期。⑤RISC 机器选用精简指令系统，适合于流水线工作，大多数指令可在一个时钟周期内完成。

(3) 便于设计，可降低成本，提高可靠性。RISC 指令系统简单，故机器设计周期短，逻辑简单，设计出错可能性小，有错也容易发现，可靠性高。

(4) 有效支持高级语言程序。由于 RISC 指令系统指令少，寻址方式少，因此编译程序可选择更有效的指令和寻址方式。而且 RISC 机器的通用寄存器多，可尽量安排寄存器的操作，提高了编译程序的代码优化效率，以更有效地支持高级语言程序。

当然，和 CISC 结构相比较，尽管 RISC 结构有上述的优点，但决不能认为 RISC 结构就可以完全取代 CISC 结构。事实上，RISC 和 CISC 各有优势，而且界限并不明显。有些典型的 CISC 处理器中内部加入了 RISC 的特性，如 Intel 公司的 80486、Pentium 系列的 CPU 就融合了 RISC 和 CISC 的优势。

3.4.2 典型指令集架构

指令集架构是指一种类型的 CPU 中用来计算和控制计算机系统的一套指令的集合。指令集架构主要规定了指令格式、数据类型及格式、寻址方式和可访问地址空间的大小、程序可访问的寄存器个数和位数、存储保护方式等。指令集通常包括三大类主要的指令类型：运算指令、分支指令和访存指令。此外，还包括架构相关指令、复杂操作指令和其他特殊用途指令。因此，一种 CPU 的指令集架构不仅决定了 CPU 所要求的能力，而且也决定了指令的格式和 CPU 的结构。下面介绍的 80x86 架构、ARM 架构和 RISC-V 是典型的指令集架构。

1. 80x86 架构

80x86 架构也称为 IA-32，是一个 32 位架构，最初由 Intel 公司研发，AMD 也在销售与 80x86 兼容的微处理器。80x86 漫长而曲折的历史可以追溯到 1978 年。当时，Intel 推出了 16 位的 8086 微处理器，IBM 选择了 8086 的姐妹产品 8088 作为 IBM 第一代个人计算机的 CPU。1985 年，Intel 公司推出了 32 位的微处理器 80386。它对于 8086 是向后兼容的，可以运行和使用为早期 PC 开发的软件。兼容 80386 的处理器体系结构称为 80x86 处理器。Pentium、Core 和 Athlon 处理器都是著名的 80x86 处理器。

这些年来，Intel 和 AMD 把更多的新指令和功能都塞进了这个陈旧的体系结构中，其结果就是这种体系结构很不优雅。然而，软件的兼容性比技术的优雅性更加重要，所以 80x86 在这 20 多年内都是 PC 的事实标准。每年卖出的 80x86 处理器超过 1 亿片，巨大的市场保证了每年可用高达 50 亿美元的研究开发经费来对处理器做进一步的改进。

80x86 架构是目前唯一的主流复杂指令集，垄断了个人计算机和服务器处理器市场。80x86 架构兼容 16 位、32 位指令集，双字(4 字节)长度的存储器访问时可以不对齐存储器地址，操作数以小端次序存储。兼容性一直都是 80x86 架构发展的驱动力，但在较新的微架构中，80x86 处理器会把 80x86 指令转换为更像 RISC 的微指令再予执行，从而获得可与 RISC 比拟的超标量性能，而仍然保持向前兼容。

到2002年，由于32位特性的长度，80x86的架构开始到达某些设计的极限。Intel原本决定在64位时代完全舍弃80x86的兼容性，推出新架构的IA-64作为其Itanium处理器产品线的基础。但IA-64与80x86不兼容，使用模拟形式来运行80x86的软件不仅效率低下，还影响其他程序的运行。

AMD主动把32位的80x86架构扩充为64位的AMD64架构。由于AMD的64位处理器产品线首先进入市场，且微软也不愿意为Intel和AMD开发两套不同的64位操作系统，因此Intel被迫采纳AMD64指令集，同时增加了一些新的指令并将其扩充到它们自己的产品中，命名为EM64T架构。EM64T后来被Intel正式更名为Intel 64。

2. ARM架构

ARM架构是由ARM公司开发设计的32位精简指令集处理器架构。

ARM从1983年开始研发，1985年开发出ARM 1 Sample版，而首颗“真正”的产能型ARM 2于次年量产。截至2021年，ARM已经开发出了9代架构。ARM架构的版本与典型内核如表3-4所示。

表3-4　ARM架构的版本与典型内核

架构版本	架构分类	典型内核	说　明
ARM v9	-A	A710	ARM v9-A架构建立在ARM v8-A架构之上并向下兼容
ARM v8	-A	A32 A34 A35 A55 A65 A72 A73 A75 A76 A77	兼容64位和32位执行状态的能力。AArch64执行状态支持A64指令集；AArch32是一个32位执行状态，它保留了与ARM v7-A架构的兼容性
	-R	R52 R52+ R82	扩充了AArch64架构
	-M	M85 M55 M33 M35P M23	支持Trust Zone技术
ARM v7	-A	A5 A15	支持基于内存管理单元(MMU)的虚拟内存系统架构(VMSA)，支持ARM(A32)和Thumb(T32)指令集
	-R	R4 R5 R7 R8	支持ARM(A32)和Thumb(T32)指令集
	-M	M7 M4 M3 M1	支持Thumb(T32)指令集
ARM v6	-M	M0	ARM v6-M架构是ARM v7-M的子集合，与ARM v7-M向上兼容；支持T32指令集
		ARM 11	集成Jazelle技术，八级流水线
ARM v5		ARM 10 Xscale	应用处理器，六级流水线
ARM v4		ARM 7 ARM 8 ARM 9 StrongARM	v4版架构在v3版上作了进一步扩充，是目前应用广泛的ARM体系结构
ARM v3		ARM 6	寻址空间增至32位
ARM v2		ARM 2 ARM 3	该版架构对v1版进行了扩展，包含了对32位乘法指令和协处理器指令的支持
ARM v1		ARM 1	该版架构只在原型机ARM 1上出现过，只有26位的寻址空间，没有用于商业产品

下面简要介绍ARM指令集架构的发展。

1985年4月26日发布的ARM v1包含45条基本指令，指令长度为32位，可对两个32位操作数进行操作；配置有一个24位的程序计数器和26位地址线，可以寻址的最大地址

空间为64MB；有16个32位通用寄存器。ARM v1 ISA没有商用。

1986年发布了ARM v2 ISA，应用在ARM 2处理器上，这也是ARM架构第一款正式的商用处理器。ARM v2 ISA中一共包含56条指令，指令长度为32位，可对两个32位操作数进行操作。其中有4条移动指令(MOV，MVN)、5条载入指令(LDM，LDR)、5条存储指令(STM，STR)、14条算术运算指令、8条逻辑运算指令、8条比较指令、2条分支指令、1条软件中断指令和9条对协处理器的支持指令。ARM v2 ISA有27个32位寄存器。在之后的迭代版本ARM v2a中，增加了两条原子性存储、加载指令(SWP，SWPB)。

1989年，ARM公司发布了基于ARM v2a架构的1.5μm工艺的ARM 3处理器，该处理器相比ARM 2增加了一级64路组相联的4KB cache结构，是第一个采用了片上cache结构的处理器，带来了相当可观的性能改进。1990年，ARM公司发布了ARM v3指令集架构，程序计数器(PC)可为32位，对应的地址空间也就增加到了4GB。

1993年，ARM v4架构诞生，这个架构被广泛使用，ARM 7(7TDMI)、ARM 8、ARM 9(9TDMI)和StrongARM处理器均采用了该架构。ARM v4架构中包含17个CPU寄存器，用于保存从内存带入处理器的32位宽的数据以及由ALU计算的32位的结果。同时，ARM v4架构中增加了处理器的特权模式，使用用户模式下的寄存器。在指令中，ARM v4相比ARM v3增加了有符号、无符号的半字和有符号字节的载入和存储指令，引入了基于JTAG接口的ICE调试。ARM v4 T架构中的T表示Thumb指令集的引入。Thumb指令不仅可提高代码密度，还能使先进的ARM处理器工作在16位的环境下。

1998年，ARM v5 TEJ架构发布，提高了T变种中ARM/Thumb指令混合使用的效率；新增了E系列的指令集变种，增强了ARM处理器在数字信号处理算法上的应用效率；新增了J系列指令变种，增加了BXJ指令和支持Jazelle架构扩展所需的其他指令。

2001年，ARM v6架构诞生，它包含了上一代ARM v5 TEJ中的所有特性，并引入了混合16位/32位的Thumb-2指令集。

2004年，ARM v7架构发布。ARM v7共有3种指令集配置：ARM v7-A、ARM v7-R、ARM v7-M。ARM v7-A中的A代表Application，这种架构实现了具有多种模式的传统ARM架构，支持基于内存管理单元的虚拟内存系统架构，同时支持经典的ARM与Thumb指令集。处理器Cortex-A8/9/5/7/15/17都是基于该架构的。ARM v7-R中的R代表Real-time，支持基于内存保护单元(MPU)的受保护内存系统体系结构(Protected Memory System Architecture，PMSA)。处理器Cortex-R4/R5/R7/R8都是基于该架构的。ARM v7-M中的M代表Microcontroller，实现了一个用于低延迟中断处理的模型，使用寄存器的硬件堆栈，支持用高级语言编写的中断处理程序。处理器Cortex-M3/4/7便基于该架构。ARM v7架构还采用了NEON技术，将DSP和媒体处理能力提高了近4倍。

2011年11月，ARM v8架构诞生，这是ARM发布的第一个64位指令集架构，也是ARM公司目前主流的芯片指令集，共发布了ARM v8-A到ARM v8.6-A 7个版本。ARM v8指令集架构支持64位(AArch64)与32位(AArch32)两个执行状态。在AArch64状态下，ARM v8架构只支持A64指令集，这是一个固定长度的指令集，使用32位指令编码；地址保存在64位寄存器中，指令可以使用64位寄存器进行处理；提供了31个64位通用寄存器，其中X30寄存器用作过程链接寄存器；提供了64位PC、SP和异常链接寄存器(ELR)；提供了对64位虚拟地址的支持，并定义了新的ARM v8异常模型。在AArch32

状态下，ARM v8 架构提供了 13 个 32 位通用寄存器，1 个 32 位 PC、SP 和链接寄存器(LR)，LR 既可用作 ELR，也用作过程链接寄存器，这与之前 32 位版本的 ARM 架构兼容；为 SIMD 矢量运算操作提供了 32 个 64 位寄存器；提供了对 32 位虚拟地址的支持，同时提供了 A32 和 T32 两个指令集，A32 是一个 32 位编码定长指令集，T32 是运行在 32 位状态下的 16 位混合编码的 Thumb 指令集。

2021 年 3 月 31 日，ARM 公司推出了最新的 ARM v9 架构，兼容 ARM v8 指令集；升级了 SVE2(scalable vector extensions，可伸缩向量扩展)指令集，可以支持多倍 128 位运算，最多 2048 位；全面增强了机器学习、DSP 等方面的能力，对物联网、5G、AR/VR 等带来了更大的性能提升。另外，ARM v9 还推出了 CCA，引入了动态域技术，增强了系统安全性，不会轻易被破解和攻击。

ARM 既不制造芯片，也不销售实际的芯片给终端客户，而是通过授权其 RISC 指令集架构和处理器设计方案，由合作伙伴生产出各具特色的芯片。ARM 公司将 ARM RISC 精简指令集授权给客户；客户可以对 ARM 指令集进行大幅度改造，甚至可以对 ARM 指令集进行扩展或缩减；之后，客户根据自己改进过的指令集研发处理器架构，从而在根源上做到对处理器架构的差异化设计，保护自研芯片的知识产权，达成独特竞争力同时又兼容 ARM 的完善生态环境。

苹果的 A 系列处理器是基于 ARM 指令集架构授权自研内核的成功典范。2012 年 9 月，苹果随 iPhone5 上市发布了 A6 处理器 SoC，A6 是基于 ARM v7 架构的。2013 年 9 月，苹果率先发布了搭载基于 ARM v8 架构研发的、64 位 Cyclone 架构的、双核 A7 处理器。2020 年，苹果宣称新发布的 A14 Bionic 芯片性能已经堪比部分笔记本处理器。

ARM 处理器架构授权是指 ARM 将自行设计的处理器内核 IP 授权给客户。客户可以直接将内核 RTL(register transition level)代码在芯片前端设计时集成在芯片处理器模块中。客户也可以对处理器缓存、核数、频率进行配置，通过系统总线与其他功能模块、外设接口、主存储接口模块等连接，生成完整的芯片。

ARM Cortex 系列处理器内核是 ARM 家族中占据处理器 IP 市场的核心系列。其中，Cortex-A 系列面向高性能计算需求、运行多种操作系统和程序任务的应用领域，例如智能手机、平板电脑、机顶盒、数字电视、路由器和监控 SoC 芯片等。Cortex-A 有以 A7x 系列为代表的高性能大核产品线和以 A5x 系列为代表的低功耗小核产品线。Cortex-M 系列内核面积更小，能效比更高，覆盖智能测量、人机接口设备、汽车和工业控制系统、大型家用电器、消费性产品和医疗器械等应用需求，在全球 32 位 MCU 市场占据主导地位。Cortex-R 是面向实时应用的高性能处理器系列，运行在比较高的时钟频率，其响应延迟非常低，主要应用在硬盘控制器、汽车传动系统和无线通信的基带控制等领域。

在智能移动设备兴起的近 20 年，基于 ARM 精简指令集架构的 ARM 内核微架构 IP 选择多样，设计精简可靠，在低功耗领域表现优异。这种授权模式在以手机、平板为代表的移动终端芯片、机顶盒、视频监控等应用的媒体芯片领域获得了广泛的成功，ARM 因此也成为移动互联时代的处理器 IP 授权霸主。

3. RISC-V 架构

RISC-V 起源于 2010 年加州大学伯克利分校的 David Patterson 与 Krste Asanovic 教授研究团队准备启动的一个新项目，该项目需要选择一种处理器指令集。由于当时已有的指令集 ARM、MIPS、SPARC、80x86 存在设计越来越复杂及知识产权等问题，因此他们决定重新设计一套指令集。伯克利研究团队认为，指令集作为软硬件接口的一种说明和描述规范，不应该像 ARM、80x86 等指令集那样需要付费授权才能使用，而应该开放和免费。

RISC-V 是第五代精简指令集，是一种高质量、免许可证、开放的 RISC 指令集架构，是一套由 RISC-V 基金会维护的标准，适用于多种类型的计算系统。开源的 RISC-V 打破了 ARM、80x86 架构的技术壁垒，便于开发者进行开发，目前国内外都在流行该架构。

RISC-V 只是定义了一套指令集架构规范，没有具体实现的代码，需要开发者根据该架构手册，通过使用 VHDL 编程来具体实现基于 RISC-V 的处理器，再移植到满足 RISC-V 需求的硬件平台（处理器芯片或开发板）上去。

使用 RISC-V 架构的同时还要配套开发相应的编译器 GCC 以及调试器，如 openOCD。RISC-V 官方提供了兼容性测试代码，可以测试每一条指令的运行结果，验证开发者是否实现了 RISC-V 规范。

目前基于 RISC-V 架构的开源处理器有很多，既有标量处理器 Rocket，也有超标量处理器 BOOM，还有面向嵌入式领域的 Z-scale、PicoRV32 等。

RISC-V 采用模块化的指令集，易于扩展、组装，包括一个基本整数指令集和多个可选的扩展指令集。

由于 80x86 和 ARM 发展时间长，已经形成了强大的生态体系，因此，RISC-V 短期内难以在计算机领域和移动互联领域替代它们。智能物联网时代带来了低延时、大容量、万亿设备互联，场景丰富、万物互联、智能化将催生新的芯片市场需求，而丰富的应用场景也导致 AIoT 市场呈现碎片化和多样化，对 CPU 的需求也极为多样，现有的处理器设计并不能有效应对。RISC-V 架构以其极致精简、灵活以及模块化的特性，可以针对不同应用灵活修改指令集和芯片架构设计。此外，很多智能设备对于成本较敏感，RISC-V 架构的免费授权对于芯片厂商就显得非常有意义了。

市场调研机构 Semico Research 的研究结果显示，预计到 2025 年，采用 RISC-V 架构的芯片数量将增至 624 亿颗。同时全球顶级学府、科研机构、芯片巨头纷纷参与，各国政府出台政策支持 RISC-V 的发展和商业化，使 RISC-V 有望成为 80x86 和 ARM 之后指令集架构的第三极。

知识拓展

国产龙芯指令集架构

2021 年 3 月 31 日，龙芯中科技术有限公司发布了具有自主知识产权的龙芯指令集架构，简称 LoongArch 架构。LoongArch 具有 RISC 指令架构的典型特征，采用 Load/store 结构，分为 32 位和 64 位两个版本，分别简称为 LA32 和 LA64。LA64 应用级向下二进制兼容 LA32。所谓“应用级向下二进制兼容”，一方面是指采用 LA32 应用软件的二进制可以

直接运行在兼容 LA 架构的机器上并获得相同的运行结果，另一方面是指这种向下二进制兼容仅限于应用软件。

LoongArch 架构采用基础部分(Loongson base)加扩展部分的组织形式，如图 3-25 所示。其中扩展部分包括二进制翻译扩展(Loongson binary translation，LBT)、虚拟化扩展(Loongson virtualization，LVZ)、向量扩展(Loongson SIMD extension，LSX)和高级向量扩展(Loongson advanced SIMD extension，LASX)。LoongArch 是全新的指令集，包含基础指令 337 条、二进制翻译扩展指令 176 条、虚拟机扩展指令 10 条、向量扩展指令 1024 条、高级向量扩展指令 1018 条，共计 2565 条原生指令。

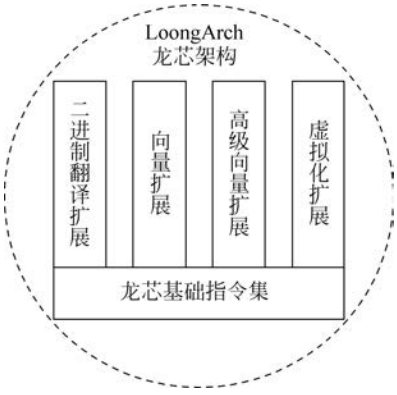

图 3-25　LoongArch 架构示意图

在 LoongArch 架构中，所有指令均采用 32 位固定长度，且指令的地址都要求 4 字节边界对齐。该架构提供了 32 个通用寄存器和 32 个浮点/向量寄存器，包含 3 种无立即数格式和 6 种有立即数格式。LoongArch 的指令系统在设计时以先进性、扩展性、兼容性为目标，其中兼容性是指融合 MIPS、80x86、ARM 指令系统的主要特点，支持高效的二进制翻译。

龙芯提供的基于 LoongArch 的 Linux 操作系统除了可运行原生的 LoongArch 程序外，还能通过翻译的方式兼容 MIPS、80x86、ARM、RISC-V 这几种指令集的 Linux 程序。LoongArch 对 MIPS 指令的翻译效率是 100%，对 ARM 可以达到 90%。最难的当属 80x86，在 Linux 下翻译的效率可达 80%，Windows 下的效率还要降低到 70%，但后续会有更多的优化。

3.5　指令系统举例

前面几节主要介绍了指令的格式、寻址方式、指令的操作类型以及指令集架构。本节通过对几种典型指令系统的实例分析，加深读者对指令系统的理解。

3.5.1　Intel 80x86 指令系统

Intel 80x86 指令系统是从 16 位的 8086/8088 指令系统发展而来的。为了满足兼容性，后续系统的指令集在其基础上增加了一些指令，如有些指令的操作数可以是 32 位，有些指令的功能得到了扩充，指令功能变得越来越复杂，指令条数越来越多。所以，Intel 80x86 指令系统是典型的 CISC 指令系统。

1. 寄存器结构

Intel 80x86 指令系统使用的寄存器主要包括通用寄存器、专用寄存器和段寄存器，其结构如图 3-26 所示。

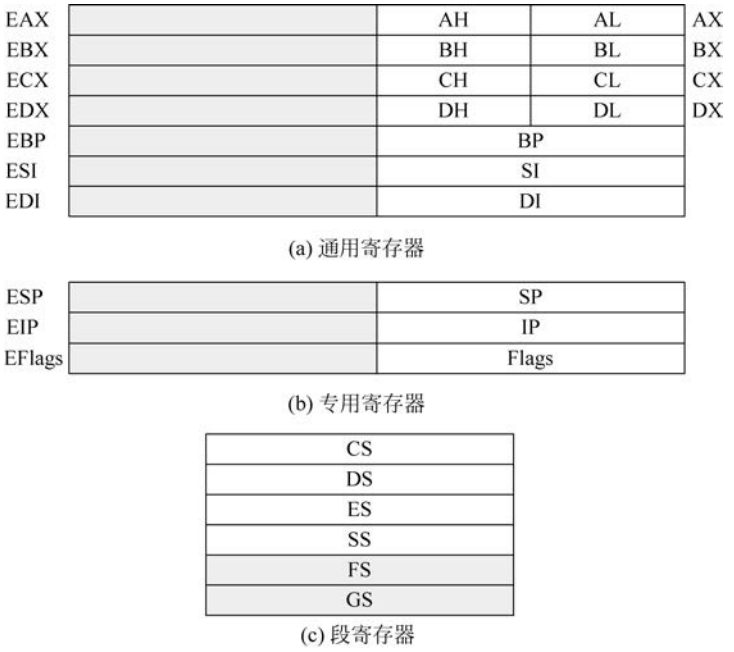

图 3-26 Intel 80x86 指令系统的寄存器结构

对寄存器 EAX、EBX、ECX、EDX、EBP、ESI 和 EDI 可以 32 位或对其低 16 位的形式被访问，其中 EAX、EBX、ECX、EDX 的低 16 位还可以以字节形式被访问。当这些寄存器以字或字节形式被访问时，不被访问的其他部分不受影响。ESP、EIP 和 EFlags 可以作为 32 位的寄存器使用，也可使用其低 16 位。

2. 数据类型

操作数是指令的操作对象。在 Intel 80x86 指令系统中，可处理的数据类型的长度有字节、字、双字、四字，还有一些特殊的数据类型，如表 3-5 所示。

表 3-5 Intel 80x86 指令系统可处理的数据类型

数据类型	说　明
无符号数	包括 8 位/16 位/32 位无符号数，分别对应于字节、字、双字
有符号数	以补码形式表示，其最高位是符号位，其余位是数值位，支持 8 位、16 位、32 位、64 位(浮点单元)有符号整数
浮点数	包括单精度实数、双精度实数、扩展精度实数
BCD 数	包括组合的 BCD 数据(每个字节包含两位十进制数)和非组合的 BCD 数据(每个字节只含一位十进制数，高四位为 0)
串数据	串数据是一串连续的数据，可由字节组成，也可由字或双字组成
ASCII	一种标准的单字节字符编码方案，用于基于文本的数据
指针数据	指针数据指出给定数据在存储器中的地址，包括：近程指针——32 位逻辑地址，即 32 位段内偏移量，用于段内数据访问或段内转移；远程指针——48 位逻辑地址，即由 16 位段选择符和 32 位偏移量组成，用于段间数据访问或段间转移

3. 指令格式

Intel 80x86 指令集的指令长度一般在 1～12 字节间变化，根据需要，可以包括一个或多个前缀(用于限定指令操作时的属性)，这种变长指令格式是典型的 CISC 结构特征。Intel 80x86 指令集的变长指令格式一是为了满足兼容性，二是希望能给编译器的编写者提供更有效的代码，其指令格式如图 3-27 所示。

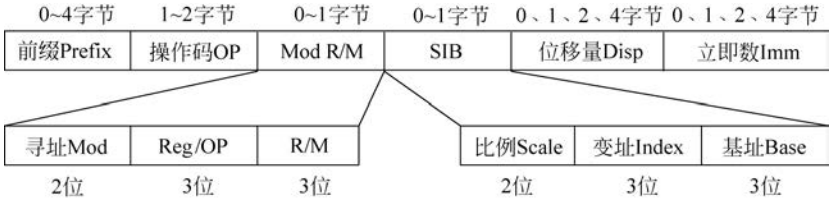

图 3-27　Intel 80x86 的指令格式

各个字段含义如下。

(1) 前缀 Prefix。前缀 Prefix 包括指令前缀、段前缀、操作数大小和地址大小 4 个域，这 4 个域都是可选项，长度均为 0～1 字节，作用是对其后的指令本身进行约定。前缀并不改变执行的内容，但使用前缀之后，指令就有了新的属性。

指令前缀包括 LOCK(锁定)前缀和重复前缀。该域编码为 F0H 时，表示 LOCK(锁定)前缀，用于多处理器环境中共享存储器的排他性访问；编码为 F2H 时，表示指令重复前缀 REPNE/REPNZ，其功能是当 ECX 不等于零且两数不相等时重复执行指令；编码为 F3H 时，表示指令重复前缀 REP/REPE/REPZ，其中 REP 的功能是当 ECX 不等于零时重复执行指令，REPE/REPZ 的功能是当 ECX 不等于零且两数相等时重复执行指令。重复前缀用于字符串的重复操作，以获得比软件循环方法更快的速度。

段前缀用来指定使用哪个段寄存器取代默认的段寄存器。6 个段寄存器 CS、SS、DS、ES、FS、GS 对应的段前缀编码分别为 2EH、36H、3EH、26H、64H、65H。例如 MOV BX，[SI+100H]指令的默认段寄存器为 DS。如果需要修改其段寄存器为 ES，则可以将指令改写为 MOV BX，ES:[SI+100H]，该指令的机器码最终会形成 26H 的段前缀。

对于操作数大小和地址大小，在实地址模式下，操作数和地址的默认长度是 16 位。在保护模式下，由段描述符中的 D 位来确定默认长度，若 D=1，操作数和地址的默认长度是 32 位；若 D=0，二者的默认长度是 16 位。

(2) 操作码 OP。操作码 OP 长度为 1～ 2 字节。该字段除包括真正的指令操作码 OP 外，还可能包括操作数以及控制信息，具体格式如图 3-28 所示。

图 3-28(a)所示为单地址指令，其中操作码字段为 5 位，剩余 3 位表示通用寄存器编号。常见指令有 PUSH、POP、DEC、INC 等。

图 3-28(b)所示为两地址指令，其中操作码字段为 6 位。剩余 2 位中，d=1，表示后续 Mod R/M 字段中的寄存器操作数为目的寄存器，否则为源寄存器；而 w 位表示操作数位宽，w=0 时表示 8 位的字节操作数，否则表示 16 位或 32 位操作数。操作数位宽取决于运

图 3-28　Intel 80x86 指令操作码的字段格式

行环境和指令前缀。

图 3-28(c)所示的操作码字段为 8 位，可包含零地址、一地址、二地址指令。

图 3-28(d)所示的操作码字段为双字节，高字节编码为 0FH，低字节为扩展操作码部分。

注意：指令字中是否包含后续的 Mod R/M、SIB、Disp、Imm 字段完全取决于操作码字段的值。不同的操作码对应不同的寻址方式，所需字段也不完全相同。

(3) Mod R/M。Mod R/M 字段长度为 1 字节，具体划分如图 3-27 所示。该字段通常用于描述操作数及其寻址方式。80x86 指令集规定双操作数最多只能有一个内存操作数，其中 3 位 Reg/OP 域表示操作数为寄存器寻址方式的寄存器编号，该操作数是源操作数还是目的操作数由操作码字段的 d 位决定。对于单地址指令，Mod R/M 字段作为指令的扩展操作码字段。

另外一个操作数由 3 位的 R/M 域表示。R/M 的意思就是该操作数有可能是寄存器操作数，也有可能是内存操作数。其寻址方式由 2 位的 Mod 域决定，当 Mod＝11 时表示寄存器寻址，具体的寻址方式还依赖于指令字中后续的 SIB、Disp 字段。

(4) SIB。SIB 字段长度为 1 字节，具体划分如图 3-27 所示，它只作为 32 位寻址方式(基地址＋比例×变址＋Disp)的补充。在 SIB 字段中，2 位的比例 Scale 域可能的取值为 00、01、10、11 四种情况之一，对应的比例值分别为 1、2、4、8；3 位的变址 Index 域用于指定变址寄存器编号；3 位的基址 Base 域是基址域，用于指定基址寄存器编号。

(5) 位移量 Disp。位移量 Disp 是一个相对于指令指针的带符号数，在指令中直接给出寻址方式所需的位移量。当该字段为存储器寻址时，它是一个计算有效地址或程序相对地址的常量。根据 w 位以及当前操作模式 16 位/32 位，位移量可以是 1、2、4 字节。一般位移量有三种形式：有符号相对位移量、无符号相对位移量和绝对位移量。

(6) 立即数 Imm。立即数 Imm 通常是在立即寻址方式指令中直接给出的操作数，其长度是 0～4 字节。如果编码中有这个字段，则包含一个被用作指令操作数或自变量的常量。

4. 寻址方式

Intel 80x86 指令系统主要有 9 种寻址方式，其中有 7 种可实现对存储器操作数寻址，具体如表 3-6 所示。

表 3-6　Intel 80x86 的寻址方式

序号	寻址方式名称	有效地址 EA 计算	说　　明
(1)	立即寻址		操作数在指令中
(2)	寄存器寻址		操作数在某寄存器中，指令给出寄存器号
(3)	直接寻址	EA＝Disp	Disp 为偏移量
(4)	基址寻址	EA＝(B)	B 为基址寄存器
(5)	基址＋偏移量	EA＝(B)＋Disp	
(6)	比例变址＋偏移量	EA＝(I) * S＋Disp	I 为变址寄存器，S 为比例因子
(7)	基址＋变址＋偏移	EA＝(B)＋(I)＋Disp	
(8)	基址＋比例变址＋偏移量	EA＝(B)＋(I) * S＋Disp	
(9)	相对寻址	指令地址＝(PC)＋Disp	PC 为程序计数器或当前指令指针寄存器

下面对 32 位寻址方式进行说明。

(1) 立即寻址。立即数可以是 8 位、16 位、32 位。

(2) 寄存器地址。一般指令或使用 8 位通用寄存器，或使用 16 位通用寄存器，或使用 32 位通用寄存器。对 64 位浮点数操作，要使用一对 32 位寄存器。少数指令以段寄存器来实现寄存器寻址方式。

(3) 直接寻址。也称偏移量寻址方式，偏移量长度可以是 8 位、16 位、32 位。

(4) 基址寻址。基址寄存器 B 可以是上述通用寄存器中的任何一个。基址寄存器 B 的内容为有效地址。

(5) 基址＋偏移量寻址。基址寄存器 B 是 32 位通用寄存器中的任何一个。

(6) 比例地址＋偏移量寻址。也称为变址寻址方式，变址寄存器 I 是 32 位通用寄存器中除 ESP 外的任何一个，而且可将此变址寄存器内容乘以 1、2、4 或 8 的比例因子 S，然后再加上偏移量而得到有效地址。

(7)、(8) 两种寻址方式是(4)、(6)两种寻址方式的组合，此时偏移量可有可无。

(9) 相对寻址：适用于转移控制类指令。用当前指令指针寄存器 EIP 或 IP 的内容(下一条指令地址)加上一个有符号的偏移量，形成 CS 段的段内偏移。

5. 操作类型

Intel 80x86 指令系统是一种典型的 CISC 指令，全部指令可分为整型类指令、浮点类指令和特权指令。

整型类指令用于完成对整数的操作，具体又可以分为若干组：数据传送、二/十进制算术运算、逻辑操作、循环移位操作、字符串操作、位测试与字节操作、控制转移操作、标志控制、段寄存器以及其他指令。

浮点类指令是由浮点部件(FPU)所执行的指令，用来对浮点数、扩展的整型数、二进制编码的十进制(BCD)数等操作数进行操作。

特权指令用于实现系统级的功能，如加载系统级寄存器(GDTR、IDTR、LDTR 等)，管理高速缓存，管理中断或设置调试寄存器等，具体的操作类型如表 3-7 所示。

表 3-7 Intel 80x86 指令系统的操作类型

类 型	指 令	操 作 说 明
数据传送	MOV	在寄存器和存储器之间或寄存器之间传送数据
	PUSH/POP	将数据进栈/出栈
	PUSHA	将所有通用寄存器的内容压入堆栈
	MOVSX	传送字节、字、双字,符号会被扩展
	LEA	装入有效地址
	XLAT	代码转换
	IN/OUT	输入输出
算术运算	ADD	加法
	SUB	减法
	MUL	乘法
	IDIV	除法
逻辑运算	AND	逻辑与
	BTS	位测试并置位
	BSF	扫描一个字节或一个字
	SHL/SHR	逻辑左移/逻辑右移
	SAL/SAR	算术左移/算术右移
	ROL/ROR	循环左移/循环右移
	SET_{CC}	根据状态标志定义的 16 个条件中的一个
控制转移	JMP	无条件转移
	CALL	转子程序,并把断点压入堆栈
	JE/JZ	如果相等则转移
	LOOPE/LOOPZ	如果相等则循环
	INT/INTO	中断指令
串操作	MOVS	传送字节、字、双字的串
	LODS	加载字节、字、双字的串
高级语言	ENTER	生成一个可以用来实现块结构高级语言规则的栈
支持	LEAVE	为 ENTER 的逆
	BOUND	检查阵列的边界
标志控制	STC	置进位标志
	LAHF	将标志加载到 AH 寄存器
段寄存器	LDS/LES/LSSLFS/LGS	将指针送寄存器和 DS、ES、SS、FS、GS
	HLT	保持原状态
	LOCK	封锁总线,Pentium 可独占存储器
系统控制	ESC	指示后面的指令支持高精度整数和浮点数计算
	WAIT	处理器处于等待状态,直到 BUSY 信号撤除
保护	SGDT	保存全局描述符表
	LSI	将段限装入一个用户指定的寄存器中
	VERR/VERW	读/写核查段
	INVD	将内部 cache 清空
	WBINVD	将更改行写回寄存器后将内部 cache 清空
	INVLPG	使一个转换后的 TLB 无效

3.5.2　RISC-V 指令系统

RISC-V 是第五代 RISC 指令系统，吸收了 ARM、MIPS 等 RISC 指令系统的优点。RISC-V 是重新开发设计的一套开源指令系统，目前由非营利的 RISC-V 基金会负责运营维护。

RISC-V 采用模块化的指令集，易于扩展、组装，包括一个基本整数指令集和多个可选的扩展指令集，其中唯一强制要求实现的是一个基本指令集，允许在实现中以可选的形式实现其他标准化和非标准化的指令集扩展。RISC-V 的指令集模块如表 3-8 所示。模块化的指令集方便进行灵活定制与扩展，既可用于嵌入式 MCU，也适合构造服务器、家用电器、工控控制以及传感器中的 CPU。

表 3-8　RISC-V 的指令集模块

指令集	名称	指令数	说　明
基本指令集	RV32I	47	整数指令，包含算术、分支、访存；32 位寻址空间，32 个 32 位寄存器
	RV32E	47	指令与 RV32I 一样，只是寄存器数量变为 16 个，用于嵌入式环境
	RV64I	59	整数指令，64 位寻址空间，32 个 64 位寄存器
	RV128I	71	整数指令，128 位寻址空间，32 个 128 位寄存器
扩展指令集	M	8	包含 4 条乘法、2 条除法、2 条余数操作指令
	A	11	包含原子操作指令，比如读-修改-写、比较-交换等
	F	26	包含单精度浮点指令
	D	26	包含双精度浮点指令
	Q	26	包含四倍精度浮点指令
	C	46	压缩指令集，其中的指令长度是 16 位，主要目的是减少代码大小

注：特定组合 IMAFD 被称为“通用(general)”组合，用英文字母 G 表示。其他的扩展指令集还包括 B 标准扩展(位操作)、E 标准扩展(嵌入式)、H 特权态架构扩展(支持管理程序(hypervisor))、J 标准扩展(动态翻译语言)、L 标准扩展(十进制浮点)、N 标准扩展(用户态中断)、P 标准扩展(封装的单指令多数据(packed-SIMD)指令)、Q 标准扩展(四精度浮点)。

本小节主要探讨 RISC-V 指令系统中的 RV32I 模块。RV32I 指令集有 47 条指令，能够满足现代操作系统运行的基本要求。

1. RISC-V 通用寄存器

RISC-V 体系结构中包含 32 个 32 位的通用寄存器，在汇编语言中可以用 x0～x31 表示；也可以使用寄存器的名称表示，例如 sp、tp、ra 等表示，用 5 位二进制表示寄存器的标号。RISC-V 寄存器的功能说明如表 3-9 所示。

由表 3-9 可知，RISC-V 包括恒零寄存器、线程寄存器 tp、7 个临时寄存器 t0～t6、12 个通用寄存器 s0～s11、8 个用于存储子程序参数和返回值的寄存器 a0～a7。RISC-V 和 MIPS 寄存器大同小异。

表 3-9 RISC-V 寄存器的功能说明

编号	助记符	英文全称	功能描述
x0	zero	zero	恒零值，可用 0 号寄存器参与的加法指令实现 MOV 指令
x1	ra	Return Address	返回地址
x2	sp	Stack Pointer	栈指针，指向栈顶
x3	gp	Global Pointer	全局指针
x4	tp	Thread Pointer	线程寄存器
x5～x7	t0～t2	Temporaies	临时变量，由调用者保存寄存器
x8	s0/fp	Saved Register/Frame Pointer	通用寄存器，由被调用者保存寄存器。在子程序中使用时必须先压栈保存原值，使用后应出栈恢复原值
x9	s1	Saved Registers	通用寄存器，被调用者保存寄存器
x10、x11	a0、a1	Arguments/Return values	用于存储子程序的参数或返回值
x12～x17	a2～a7	Argurments	用于存储子程序的参数
x18～x27	s2～s11	Saved Registers	通用寄存器，由被调用者保存寄存器
x28～x31	t3～t6	Temporaies	临时变量

2. RISC-V 指令格式

RISC-V 为定长指令集。由于其操作码字段预留了扩展空间，因此可以扩展为变长指令，但指令字长必须是双字节对齐。RISC-V 包括 6 种指令格式，具体如图 3-29 所示。RISC-V 指令没有寻址方式字段，寻址方式由操作码决定。RISC-V 强调的是指令硬件更容易实现，其最大特色是指令字中的各字段位置固定，这将有效减少指令译码电路中需要的多路选择器，也可提高指令译码的速度。

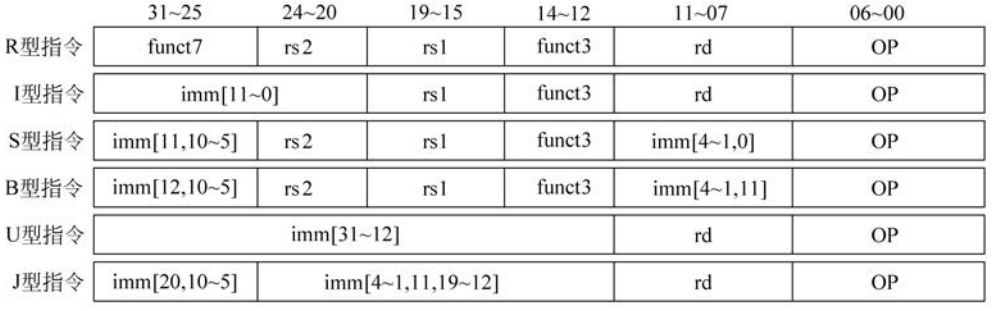

图 3-29 RISC-V 指令格式

图 3-29 中 7 位的主操作码 OP 均固定在低位，扩展操作码 funct3、funct7 字段位置也是固定的。相比 MIPS 指令集，其编码空间更大，指令可扩展性更高。另外源寄存器 rs1、rs2 以及目的寄存器 rd 在指令字中的位置也是固定不变的。

以上字段的位置固定后，剩余的位置用于填充立即数字段 imm，这也直接导致 imm 字段看起来比较混乱，不同类型指令立即数字段的长度甚至顺序都不一致。但 imm 字段的最高位都固定在指令字的最高位，方便立即数的符号扩展。另外立即数字段中部分字段尽量追求位置固定，如 I、S、B、J 型指令的 imm[10～5]字段位置固定，S、B 型指令中的 imm[4～

1]字段位置固定。

(1) R 型指令。R 型指令包括 3 个寄存器操作数,主操作码字段 OP=33H,由 funct3 和 funct7 两个字段共 10 位组成,作为扩展操作码,用于描述 R 型指令的功能。RV32I 中共有 10 条 R 型指令,主要包括算术逻辑运算指令、关系运算指令、移位指令 3 类。

(2) I 型指令。I 型指令包括两个寄存器操作数 rsl、rd 和一个 12 位立即数操作数。除主操作码 OP 字段外,funct3 字段也作为扩展操作码,用于描述 I 型指令的功能。I 型指令主要包括立即数运算指令(算术逻辑运算指令)、关系运算指令、移位指令、访存指令、系统控制类指令和特权指令。

(3) S 型指令。写存储器指令由于不存在目的寄存器的 rd 字段,因此不能采用 I 型指令格式,只能单独设置一个 S 型指令格式。

注意:funct3 字段为扩展操作码,立即数字段应扩充到目的寄存器 rd 字段的位置。

(4) B 型指令。B 型指令用于表示条件分支指令,同样 B 型指令也不存在目的寄存器的 d 字段,其指令形式和 S 型类似,但其指令字段的第 7 位和 B 型指令略有不同,所以 B 型指令也称为 SB 型指令。注意:RISC-V 指令字采用偶数对齐,指令字长为双字节的倍数,所以这里立即数左移一位。

(5) U 型指令。I 型指令立即数最多只有 12 位,范围较小。为表示更大的立即数设置了 U 型指令,这里 U 的意思是 Upper immediate。U 型指令的立即数字段为 20 位,共包含两条指令,用于立即数加载,如下所示:

```
lui    rd,imm; R[rd] = imm<<12
auipc  rd,imm; R[rd] = PC+imm<<12
```

注意:lui 指令只能将立即数加载到高 20 位。如需要加载一个完整的 32 位立即数到寄存器中,可以利用 lui 和 addi 指令配合完成。

(6) J 型指令。J 型指令可实现无条件跳转,其立即数字段也是 20 位,所以也称为 UJ 型指令。J 型指令共包含两条,可实现子程序调用和无条件跳转。

3. RISC-V 寻址方式

RISC-V 只有 4 种寻址方式,其相对寻址的生成地址方式与其他指令略有不同,具体如表 3-10 所示。

表 3-10　RISC-V 寻址方式

序号	寻 址 方 式	有效地址 EA/操作数 S	指 令 示 例
1	立即数寻址	S=imm	Addi rd,rs1,imm
2	寄存器寻址	S=R[rs1]	add rd,rs1,rs2
3	寄存器相对寻址/基址寻址	EA=R[rs1]+imm	lw rd,imm(rs1)
4	相对寻址	EA=PC+imm<<1	beq rs1,rs2,imm

4. 操作类型

RV32I 指令集的 47 条指令按照功能可以分为如下 5 类。

(1) 整数运算指令。整数运算指令实现算术、逻辑、比较等运算。

(2) 分支转移指令。分支转移指令实现条件转移、无条件转移等运算,并且没有延迟槽。

(3) 加载存储指令。加载存储指令实现字节、半字、字的加载、存储操作,采用的都是寄存器相对寻址方式。

(4) 控制与状态寄存器访问指令。控制与状态寄存器访问指令实现对系统控制与状态寄存器的原子读-写、原子读-修改、原子读-清零等操作。

(5) 系统调用指令。系统调用指令实现系统调用、调试等功能。

本章小结

本章主要讲述了以下内容。

(1) 指令系统是一台计算机中所有机器指令的集合,它是表征一台计算机性能的重要因素。指令的格式与功能不仅直接影响机器的硬件结构,而且也影响系统软件。

(2) 指令格式是指令用二进制代码表示的结构形式,通常由操作码字段和地址码字段组成。操作码字段表征指令的操作特性与功能,而地址码字段指示操作数的地址。目前多采用二地址、一地址、零地址混合方式的指令格式。指令字长度分为单字长、半字长、双字长三种形式,高档微型机中目前多采用 32 位长度的单字长形式。

(3) 寻址方式包括指令寻址方式和数据寻址方式。形成指令地址的方式称为指令寻址方式,分为顺序寻址和跳跃寻址两种。形成操作数地址的方式称为数据寻址方式。操作数可放在专用寄存器、通用寄存器、内存和指令中。数据寻址方式分为立即寻址、直接寻址、间接寻址、寄存器寻址、寄存器间接寻址、相对寻址、变址寻址、基址寻址、段寻址、复合寻址等多种。按操作数物理位置的不同,可将其分为 RR 型和 RS 型,前者比后者执行的速度快。堆栈是一种特殊的数据寻址方式,采用“先进后出”原理。按实现方法的不同,可将堆栈分为硬堆栈和软堆栈。

(4) 不同机器有不同的指令系统,一个较完善的指令系统应当包含数据传送指令、算术运算指令、逻辑运算指令、程序控制指令、输入输出指令、字符串指令、系统控制指令。

(5) RISC 指令系统是 CISC 指令系统的改进,它的最大特点是:指令条数少;指令长度固定,指令格式和寻址种类少;只有取数/存数指令访问存储器,其余指令的操作均在寄存器之间进行。

习题

一、名词解释

1. 机器指令　　2. 操作码　　3. 操作数
4. 指令系统　　5. 有效地址　　6. 寻址方式
7. 指令字长　　8. CISC　　9. RISC

二、选择题

1. 指令系统中采用不同寻址方式的主要目的是(　　)。

A. 实现存储程序和程序控制

B. 缩短指令长度,扩大寻址空间,提高编程灵活性

C. 可以直接访问外存

D. 提供扩展操作码的可能并降低指令译码难度

2. 下列说法中不正确的是(　　)。

A. 机器语言和汇编语言都是面向机器的,它们和具体机器的指令系统密切相关

B. 指令的地址字段指出的不是地址,而是操作数本身,这种寻址方式称为立即寻址

C. 堆栈是存储器的一部分,可以通过地址访问

D. 机器中的寄存器和存储单元是统一编址的

3. 下列寄存器中,汇编语言程序员可见的是(　　)。(2021 年全国硕士研究生入学统一考试计算机学科专业试题)

Ⅰ. 指令寄存器　　Ⅱ. 微指令寄存器

Ⅲ. 基址寄存器　　Ⅳ. 标志/状态寄存器

A. 仅Ⅰ、Ⅱ　　B. 仅Ⅰ、Ⅳ　　C. 仅Ⅱ、Ⅳ　　D. 仅Ⅲ、Ⅳ

4. 某计算机采用 16 位定长指令字格式,操作码位数和寻址方式位数固定;指令系统有 48 条指令,支持直接、间接、立即、相对 4 种寻址方式。在一地址指令中,直接寻址方式的可寻址范围是(　　)。(2020 年全国硕士研究生入学统一考试计算机学科专业试题)

A. 0～255　　B. 0～1023　　C. －128～127　　D. －512～511

5. 某指令功能为 R[r2]←R[r1]＋M[R[r0]],其两个操作数分别采用寄存器寻址方式、寄存器间接寻址方式。对于下列给定的功能部件,该指令在取操作数及完成加法过程中需要用到的是(　　)。(2019 年全国硕士研究生入学统一考试计算机学科专业试题)

Ⅰ. 通用寄存器组(GPRs)　　Ⅱ. 算术逻辑单元(ALU)

Ⅲ. 存储器(Memory)　　Ⅳ. 指令译码器(ID)

A. 仅Ⅰ、Ⅱ　　B. 仅Ⅰ、Ⅱ、Ⅲ　　C. 仅Ⅱ、Ⅲ、Ⅳ　　D. 仅Ⅰ、Ⅲ、Ⅳ

6. 下列寻址方式中,最适合按下标顺序访问一维数组元素的是(　　)。(2017 年全国硕士研究生入学统一考试计算机学科专业试题)

A. 相对寻址　　B. 寄存器寻址　　C. 直接寻址　　D. 变址寻址

7. 某计算机按字节编址,指令字长固定且只有两种指令格式,其中三地址指令 29 条,二地址指令 107 条,每个地址字段为 6 位,则指令字长至少应该是(　　)。(2017 年全国硕士研究生入学统一考试计算机学科专业试题)

A. 24 位　　B. 26 位　　C. 28 位　　D. 32 位

8. 下列关于 RISC 的叙述中,错误的是(　　)。(2009 年全国硕士研究生入学统一考试计算机学科专业试题)

A. RISC 普遍采用微程序控制器

B. RISC 大多数指令在一个时钟周期内完成

C. RISC 的内部通用寄存器数量比 CISC 多

D. RISC 的指令数、寻址方式和指令格式种类比 CISC 少

三、综合题

1. 一个完备的指令系统通常应满足哪几个方面的要求?

2. 常用的指令格式有哪些?

3. 指令操作码主要有哪两种编码格式?

4. 指令操作通常分哪几类？

5. 如何减少一条指令中给出的地址数？又如何减少指令中表示一个地址码的位数？

6. 什么是立即数寻址？立即数寻址中的“立即数”一般存放在哪里？

7. 简述相对寻址、变址寻址及基址寻址的原理。这三种寻址方式有哪些相同点和不同点？

8. 堆栈寻址方式的主要用途是什么？

9. 设某指令系统指令定长 12 位，每个地址段 3 位。试设计一种方案，使该指令系统有 4 条三地址指令，8 条二地址指令，180 条一地址指令。

10. 假设某计算机指令长度为 20 位，具有双操作数、单操作数、无操作数三类指令格式，每个操作数地址规定用 6 位表示。现已设计出 m 条双操作数指令，n 条无操作数指令。

试问：

(1) 若采用定长 8 位操作码字段，则这台计算机最多可以设计出多少条单操作数指令？

(2) 若操作码字段长度不固定，则这台计算机最多可以设计出多少条单操作数指令？

11. 某微型机的指令格式如图 3-30 所示。

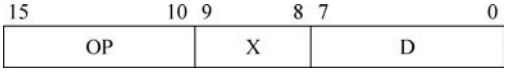

图 3-30 微型机指令格式

其中 D 为位移量，X 为寻址方式特征值。

X＝00：直接寻址；

X＝01：用变址寄存器 R_1 进行变址；

X＝10：用变址寄存器 R_2 进行变址；

X＝11：相对寻址。

设(PC)＝1234H，(R_1)＝0037H，(R_2)＝1122H(H 代表十六进制数)，请确定如下指令的有效地址：(1)4420H；(2)2244H；(3)1322H；(4)3521H；(5)6723H。

12. 某指令系统的指令定长为 16 位，指令格式如图 3-31 所示，试分析指令格式及寻址方式的特点。

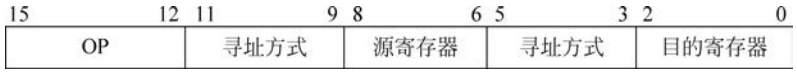

图 3-31 指令格式 1

13. 一种一地址指令格式如图 3-32 所示，其中 I 为间接特征，X 为寻址模式，D 为形式地址。I、X、D 组成该指令的操作数有效地址 EA。设 R 为变址寄存器，R_1 为基址寄存器，PC 为程序计数器，请在表 3-11 的第一列位置填入适当的寻址方式名称。

图 3-32 指令格式 2

14. 一个处理器具有如图 3-33 所示的指令格式。

表 3-11　填入寻址方式

寻址方式名称	I	X	有效地址 EA
	0	00	EA=D
	0	01	EA=(PC)+D
	0	10	EA=(R)+D
	0	11	EA=(R_1)
	1	00	EA=(D)
	1	11	EA=(((R_1)+D),D=0

6位	2位	3位	3位	
OP	X	源寄存器	目标寄存器	地址

图 3-33　指令格式 3

图 3-33 所示格式表明该处理器有 8 个通用寄存器(长度为 16 位),X 为指定的寻址模式,主存最大容量为 256KB。

(1) 假设不用通用寄存器也能直接访问主存的每一个操作数,并假设操作码域 OP=6 位,请问地址码域应该分配多少位? 指令字长度应有多少位?

(2) 假设 X=11 时,指定的那个通用寄存器用作基址寄存器,请提出一个硬件设计规则,使被指定的通用寄存器能访问 1MB 的主存空间中的每一个单元。

15. 某计算机采用 16 位定长指令字格式。假定计算机字长为 16 位,按字节编址;连接 CPU 和主存的系统总线中地址线为 20 位,数据线为 8 位。指令格式及其说明如图 3-34 所示。

格式	6位	2位	2位	2位	4位	指令功能或指令类型说明
R型	000000	rs	rt	rd	op1	R[rd] ← R[rs] op1 R[rt]
I型	op2	rs	rt	imm		含ALU运算、条件转移和访存操作3类指令
J型	op3	target				PC的低10位←target

图 3-34　指令格式 4

其中,op1～op3 为操作码,rs、rt 和 rd 为通用寄存器编号,R[r]表示寄存器 r 的内容,imm 为立即数,target 为转移目标的形式地址。请回答下列问题:

(1) ALU 的宽度是多少位? 可寻址主存空间为多少字节? 指令寄存器、主存地址寄存器(MAR)和主存数据寄存器(MDR)分别应有多少位?

(2) R 型格式最多可定义多少种操作? I 型和 J 型格式总共最多可定义多少种操作? 通用寄存器最多有多少个?

(3) 假定 op1 为 0010 和 0011 时,分别表示带符号整数减法和带符号整数乘法指令,则指令 01B2H 的功能是什么(参考上述指令功能说明的格式进行描述)? 若 1、2、3 号通用寄存器当前内容分别为 B052H、0008H、0020H,分别执行指令 01B2H 和 01B3H 后,3 号通用寄存器的内容各是什么? 各自结果是否溢出?

(4) 若采用 I 型格式的访存指令中 imm(偏移量)为带符号整数,则地址计算时应对 imm 进行零扩展还是符号扩展?

(5) 无条件转移指令可以采用上述哪种指令格式?