第3章

指 令 集

引言

指令集是计算机软件硬件的主要分界面之一，也是软件与硬件设计人员相互沟通的桥梁。无论使用哪一种程序设计语言编写的程序，无论程序的功能有多么强大、复杂，只要在计算机上运行，都需要被“翻译”和组织成一条条由 0 和 1 构成的机器语言指令。计算机的硬件设计人员利用各种手段实现指令系统；软件设计人员则用指令编制各种程序，以消除硬件指令集与人类习惯的使用方式之间的语义差异。

指令是面向芯片的语言，因此不同的计算机有不同的指令集。虽然不同指令集在表现形式、丰富性等方面存在一定差异，但它们在功能上，特别是在基本指令集的功能上都比较相似，更多的只是表现形式上的区别，就如同人类语言中的方言，虽然发音不同，但是意思是一样的。本章首先概述目前主流的 MIPS、ARM 和 x86 这 3 种指令集。然后，重点介绍 Intel x86 指令集，主要以 16 位指令集(x86-16)为例，介绍指令的基本格式、寻址方式以及不同类型指令的功能。最后，简要介绍 ARM 指令集的特点、格式以及主要指令的功能。虽然当前自动控制系统中大量采用了基于 ARM 的嵌入式技术作为控制核心，但控制程序大多会使用 C 语言等高级语言编写。因此，本章对 ARM 指令集的介绍更多地是以帮助读者对 ARM 指令集有基本认知为目标的，同时也为读者学习高级语言和汇编语言混合编程的嵌入式应用开发提供一些基础。

教学目的

- 能够描述 3 种常见指令集的特点。
- 清楚指令的一般概念、Intel 指令集和 ARM 指令集中指令的基本格式以及指令中的操作数。
- 熟悉 x86-16 指令的各种寻址方式。
- 深入理解 x86-16 指令集中常用指令的功能，包括指令操作码的含义、指令对操作数的要求、指令对标志位的影响和指令的执行结果，并在此基础上清楚 x86-32 新增指令集中主要指令的功能。
- 能够初步描述 x86-64 和 IA32 指令集的主要区别。
- 清楚 ARM 指令集的基本格式和寻址方式。
- 初步认识 ARM 的 32 位常用指令集。

3.1 计算机的语言——指令

为了帮助初次接触计算机硬件系统的读者能够更好地理解本章内容，在正式开始讲述指令之前，需要再次声明：本章描述的指令本质上是指与高低电平对应的、用 0 和 1 表示的机器语言指令，尽管这些指令都会用便于人类记忆的字母符号（称为助记符）表示，但它们绝非各种高级语言中的语句。例如，如下 C 语言中的一条简单算术运算语句：


指令系统基本概念

```
sum=x*y-z;
```

可以对应 4 条用助记符表示的指令[①]：

```
mov ax,x
mul y                    ;相当于 x*y 运算
sub ax,z
mov sum,ax
```

要使计算机能够按照人的指挥完成操作，就必须使用计算机语言。计算机语言中的基本单词称为**指令**（instruction），它是指控制计算机完成指定操作、并能够被计算机识别的命令。这句话中隐含着两层含意。“控制计算机完成指定操作”的主体是人。所以，指令的第一层含义是能够被人识别。由于执行指令的工作是由计算机中的处理器（CPU）实现的，“能够被计算机识别”就是能够被处理器识别。所以，指令的第二层含义是不同的处理器能够识别的指令可能不同。

一个处理器能够识别的全部指令称为该处理器的**指令集**（instruction set），它支持的指令集和指令的字节级编码被称为它的指令集体系结构（Instruction Set Architecture，ISA）。不同的处理器“家族”，如本章要介绍的 Intel x86-系列处理器和 ARM 处理器，有各自不同的 ISA。指令集定义了计算机硬件能完成的基本操作，其功能的强弱在一定程度上决定了硬件系统性能的高低。

◇3.1.1 常见指令集概述

无论是早期的计算机还是现代计算机，都是基于基本原理相似的硬件技术构建的。同时，所有的计算机都必须能够提供算术运算等一些基本的操作。这使得不同计算机的机器语言非常类似。因此，只要理解了一种机器语言，其他类似的机器语言也就很容易理解了。

自 20 世纪 70 年代以来，比较流行的指令集主要有 3 种。

1. MIPS 指令集

MIPS（Million Instructions Per Second）是微处理器每秒执行的百万条机器语言指令数的缩写，也是衡量微处理器速度的一个重要指标。MIPS 架构最早由斯坦福大学研制，采用精简指令集计算机（RISC）设计，其核心思想是通过简化指令使计算机的结构更加简单、合

① 一条助记符指令对应一条用 0 和 1 表示的机器语言指令。

理，从而提高 CPU 的运算速度。MIPS 指令集的主要特点是所有指令的格式和周期一致，并且采用流水线技术。这类指令集主要应用于中高端服务器和各类嵌入式系统开发。2018 年，MIPS 技术公司宣布将 MIPS 指令集开源。

2. ARM 指令集

ARM（Advanced RISC Machines）既是对一类微处理器的通称，也是一种技术的名字。ARM 体系结构目前被公认为业界最领先的 32 位和 64 位嵌入式微处理器结构，2011 年有超过 90 亿个各类设备使用 ARM 处理器，并以每年 20 亿的数量在增长。ARM 指令集也成为嵌入式领域中最流行的指令集。本章将在 3.5 节对 ARM 指令集做进一步介绍。

无论是 MIPS 指令集还是 ARM 指令集，都是基于精简指令原理设计的，其设计理念以简化单条指令功能、提高单条指令执行速度为主要目标。还有的设计者趋向于设计单条指令功能更强大的指令集，目的是减少程序需要执行的指令条数，尽管这有可能会增加程序执行的时间[①]。Intel x86 指令集就属于这种指令集。

3. Intel x86 指令集

一个指令系统在设计时通常要考虑的主要问题有指令种类的丰富性（只有加法指令的计算机不会有多大意义）、指令的执行速度（指令执行需要的时钟周期数和每个时钟周期的时间）、指令的兼容性（这是计算机系统的生命力所在）以及指令格式上的规整性等。

与 MIPS 指令集和 ARM 指令集不同的是，x86 指令集属于复杂指令集计算机（CISC），其设计目标是尽可能增强每一条指令的功能，将一些原来用软件实现的、常用的功能变成用硬件指令实现。Intel x86 指令集历经 40 余年的不断发展和完善，成为今天在个人计算机领域和云计算领域占有统治地位的指令系统。以下是 x86 指令集发展的主要里程碑：

- 1978 年，Intel 公司在上一代 8 位微处理器（如 8080）的基础上，发布了 16 位微处理器 8086/8088。
- 1980 年，Intel 8087 浮点协处理器发布。这个体系结构在 8086/8088 的基础上增加了 60 条浮点指令。
- 1982 年，Intel 公司在 8086 的基础上把地址空间扩展到 24 位，并设计了内存映射和内存保护模式。
- 1985 年，80386 微处理器在 80286 体系结构基础上将地址空间扩展到 32 位。除了 32 位的寄存器和 32 位的地址空间、加强了 8086 部分指令的功能之外，80386 还增加了对 32 位数的操作及一些新的寻址模式。
- 1989—1995 年，Intel 公司先后发布了 80486、Pentium 微处理器和 Pentium Pro 微处理器。这些微处理器均以获得更高性能为目的，在用户可见的指令集中增加了 4 条指令，分别用于支持多处理技术和条件传送。
- 1997 年，Intel 公司推出了多媒体指令增强技术（MMX），新增了 57 条指令，使用浮点栈加速多媒体和通信应用程序，通过传统的单指令流多数据流（SIMD）的方式一次处理多个短数据元素。

① 以增强指令功能为主要设计目标的指令系统称为复杂指令集计算机（CISC），以简化指令功能为主要目的称为精简指令集计算机（RISC）。CISC 和 RISC 是指令系统设计的两个不同方向。

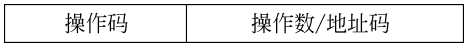
- 1999 年，Intel 公司增加了 70 条指令，将 SSE（Streaming SIMD Extension，流式 SIMO 扩展）作为 Pentium 的一部分。主要的变化是添加了 8 个独立的寄存器，把它们的长度增加到 128 位，并且增加了一个单精度浮点数据类型。这样，就可以并行进行 4 个 32 位浮点操作。为了改进内存性能，SSE 还增加了包括高速缓存的预取指令以及可以绕过缓冲器直接写内存的流存储指令。
- 2001 年，Intel 公司增加了 144 条指令，命名为 SSE2，可实现 64 位双精度浮点型数据的并行运算。这种改进不仅允许更多的多媒体操作，而且大大增强了 Pentium 4 微处理器的浮点性能。
- 2003 年，AMD 公司[①]对 x86 体系结构进行了改进，把地址空间从 32 位增加到 64 位。即将所有寄存器都拓宽到 64 位，而且寄存器的数目增加到了 16 个。指令集体系结构（ISA）的主要变化是用 64 位的地址和数据重新定义所有 x86 指令的执行（长模式）。
- 2004 年，Intel 公司增加了 128 位的原子比较和交换指令，同时发布了新一代谜题扩展指令，添加了 13 条支持复杂算术运算的指令。
- 2006—2007 年，Intel 公司先后发布了总计 224 条新指令，用于支持绝对差求和、数组结构点积计算、序列中非零数目统计以及虚拟机。在新指令中，还为 46 条基本指令集中的指令增加了 MIPS 指令等 3 操作数指令。
- 2011 年，Intel 公司发布了高级向量扩展，重新定义了 250 条指令并新增了 128 条指令。

上述发展历程说明了“兼容”的重要性，微处理器的发展、体系结构的改变都不允许对已有软件产生危害，这样才能保证在低版本上编写的程序在更高级的版本上依然能够正常运行。

虽然目前 x86 芯片的年产量相对于 ARM 芯片要少很多，但 x86 指令集在个人计算机系统中占据着绝对垄断的地位，也一直对个人计算机的更新换代起着很大的推动作用。3.3 节讨论的指令集是 x86 的 16 位指令子集，而不是整个 16 位、32 位和 64 位指令集。以下如无特殊说明，本章所述 Intel 指令集均特指 x86 的 16 位指令子集，简称 x86-16 指令集或 8086 指令集[②]。

◇3.1.2 计算机中的指令表示

计算机中的指令通常由两部分组成：一部分是操作码（OPeration Code，OPC），用便于记忆的助记符（英文单词缩写）表示；另一部分是指令操作的对象，称为操作数（operand）或地址码（address code）。指令的基本格式如图 3-1 所示。



◆图 3-1 指令的基本格式

① AMD 公司即美国超威半导体公司，主要生产各种芯片，如 CPU、GPU、APU、控制芯片组、电视卡芯片等。

② Intel x86 的 16 位指令子集主要指 8086、8088、80286 指令集。其中，8086 与 8088 微处理器有完全一样的指令集。为简单起见，以下统称为 x86-16 指令集或 8086 指令集。

操作码也称为指令码，表示指令要进行的操作类别，如加法、乘法等，是指令中必须给出的内容。指令中的**操作数**是指令执行操作所需要的数据的来源，其本质的含义就是指令执行对象的实际存放地址，所以也称为地址码。但在形式上，操作数有时能够以常数的形式出现（见 3.1.3 节），这也是本书没有使用地址码而是使用操作数这个名词的原因。

在 x86 的 16 位指令集中，指令的操作数通常有两个，分别称为源操作数和目标操作数，如图 3-2 所示。

◆图 3-2　双操作数指令格式

（1）**源操作数**（OPs）表示指令执行对象中某个对象的存放地址或执行对象本身（直接给出运算的数据）。简单地说，源操作数表示指令运算数据的来源（数据本身或存放数据的地址）。源操作数的一大特点是指令执行后不会对源操作数造成破坏。

（2）**目标操作数**（OPd）总体上可以有两层含义。在不同的指令中，目标操作数可以是指令执行的另一个对象的存放地址（即另一个运算数据）；但在所有指令中，目标操作数都一定表示指令执行结果的存放处。所以，目标操作数一定是以地址的形式出现的，并且指令执行后会对目标操作数产生影响，即用运算结果覆盖其原来的值。

指令是计算机的语言，不同系统的指令集在格式上有一定的差异。在 x86 指令集中，一条指令可以有 3 个操作数、2 个操作数、1 个操作数以及没有操作数[①]。而 ARM 指令集则允许多个操作数。由此，指令在形式上有以下 4 种：

- **零操作数指令**。指令在形式上只有操作码，操作数为隐含存在。这类指令操作的对象通常为微处理器本身。
- **单操作数指令**。指令中仅给出一个操作数。事实上，这类指令常常隐含存在另一个操作数。
- **双操作数指令**。x86 指令集中多数指令为图 3-2 所示的双操作数指令，即指令的两个操作数都显式给出。
- **三操作数指令**。从 2007 年起，Intel 公司的基本指令集中开始支持具有 3 个操作数的指令，3 个操作数中包括两个源操作数和一个目标操作数，如图 3-3 所示。由于这里的目标操作数仅用于存放执行的结果，所以称为目标地址。两个源操作数则表示运算的对象。这种指令的执行不会破坏任何一个操作数。ARM 处理器的指令集就沿用了这种指令格式（详见 3.5 节）。

◆图 3-3　三操作数指令格式

鉴于本章主要讨论 x86 处理器的 16 位指令子集，因此，以下介绍的 x86 指令仅有零操作数、单操作数和双操作数 3 种，不涉及三操作数指令。

① 事实上，每一条指令都一定有操作对象。所谓没有操作数，只是表示没有将操作对象显式给出，而是隐含给出的。

◇3.1.3 指令中的操作数

X86-16 指令中的操作数主要有 3 种类型：立即数操作数、寄存器操作数和内存操作数。以下利用将在 3.3.1 节中介绍的数据传输指令 MOV，说明这 3 种主要操作数的表现方式。对它们更深入的理解需要在学习完 3.2 节的寻址方式后才能实现。

1. 立即数操作数

立即数是指令中直接给出的运算数据，具有固定的数值(常数)，它不因指令的执行而发生变化。在 x86-16 指令集中，立即数的字长可以是 8 位或 16 位无符号数或有符号数。如果数的取值超出了字长规定的范围，就会发生错误。

特别需要注意的是：由于目标操作数是指令执行结果的存放处，而立即数是一个常数，因此立即数操作数在指令中只能作为源操作数，而不能作为目标操作数。

例如：

```
MOV AX,100
```

在这条指令中，源操作数是常数 100。这条指令的执行结果是将常数(立即数)100 写入累加器 AX。

2. 寄存器操作数

寄存器操作数主要指放在通用寄存器中的操作数。寄存器操作数既可以作为源操作数，也可以作为目标操作数。有时候，段寄存器也可以作为指令中的寄存器操作数。但由于段寄存器中存放的是逻辑段的段地址，随意修改段地址容易引起执行错误，特别是代码段的段地址更不允许指令随意修改，因此段寄存器在指令中出现的频率通常比较低。

指令指针指向的是 CPU 下一条要读取的指令在内存中的地址，因此其与段寄存器一样不允许随意修改。FLAGS 寄存器中保存着当前程序执行的现场，除个别指令外，一般也不作为操作数出现在指令中。

例如：

```
MOV AX, BX
```

在这条指令中，源操作数和目标操作数都是通用寄存器。这条指令的执行结果是将 BX 的值写入累加器 AX，执行后 AX 中原来的值将被修改，而 BX 的值不变，即该指令执行后 AX＝BX。

3. 内存操作数

内存操作数的含义是指参加运算的数据存放在内存中，用[ ]表示。其基本格式为

```
段寄存器：[偏移地址]
```

其中，“段寄存器：”表示数据所在逻辑段的段地址，在默认情况下(详见 3.2 节)可以省略，即可以不显式给出。[ ]中给出的是操作数所在内存单元的偏移地址。不同的寻找方式，其偏移地址的表现形式不同。但无论怎样的形式，[ ]内都一定是存放操作数的内存单元的偏移地址。这是理解 3.2 节中各种针对内存操作数的寻址方式的关键点。

由于这里仅讨论 16 位指令集，因此，指令操作数一般为 8 位或 16 为字长。虽然内存空

间很大，但作为指令中的操作数，8086 指令集中的内存操作数的字长原则上也只能是 8 位或 16 位，只有在极个别的指令中会出现 32 位字长的操作数（具体参见 3.3 节）。如何确定内存操作数在不同指令中的字长，也是学习具体指令时需要注意的一点。

例如：

```
MOV AH, [100]
```

在这条指令中，目标操作数是通用寄存器，源操作数是内存操作数（请关注内存操作数的表现形式）。这条指令的执行结果是将偏移地址为 100 的内存单元中的值写入累加器 AH，执行后 AH 中原来的值将被修改，而偏移地址为 100 的内存单元中的值不改变。

读者看到这里可能会问：偏移地址为 100 的内存单元在哪个逻辑段？段地址是多少？对于这些疑问，将在 3.2 节中找到答案。

内存操作数的偏移地址也称有效地址（Efficient Address，EA）。存储器操作数可以通过不同的寻址方式由指令给出。事实上，3.2 节中介绍的几种较复杂的寻址方式都针对的是内存操作数。

一条指令的执行通常包括取指令、指令译码、读取操作数、执行、传送结果 5 个步骤，它们需要的时间共同构成了指令执行的时间开销。读取寄存器操作数的时间最短，而读取内存操作数的时间与采用的寻址方式有关，不同的寻址方式，计算偏移地址需要的时间不同，其指令执行时间可能会相差很大。

以通用数据传送指令（MOV）为例，若 CPU 的时钟频率为 5MHz，即一个时钟周期为 0.2μs，则从寄存器到寄存器之间的传送指令的执行时间为

$$t = 2 \times 0.2\mu s = 0.4\mu s$$

立即数传送到寄存器的指令执行时间为

$$t = 4 \times 0.2\mu s = 0.8\mu s$$

而对于存储器到寄存器的字节传送，设存储器采用基址-变址寻址方式，则指令执行时间为

$$t = (8 + 8) \times 0.2\mu s = 3.2\mu s$$

指令的寻址方式

3.2 寻址方式

要高效率地开发微处理器软件，需要清楚每条指令的寻址方式。本节以 MOV 指令（数据传送指令，详见 3.3.1 节）为例，说明操作数的寻址方式。

所谓寻址方式，顾名思义是指获得操作数所在地址的方法。根据冯·诺依曼原理，程序执行前需要进入内存，故指令操作数通常来自内存或寄存器，在某些特殊情况下，也可以由指令直接给出，或采用默认方式（隐含）给出[①]。

本节讨论的寻址方式主要以 Intel x86-16 指令集为例[②]，除了针对立即数的寻址，其他寻址方式既适用于源操作数也适用于目标操作数。但为了描述上的方便，在以下示例中，如无特殊声明，讨论的对象均以源操作数为例。

① 这里暂时没有考虑对 I/O 接口的访问。

② Intel x86-16 指令集主要指 8086、8088、80286 指令集。

在8086指令集中，说明操作数所在地址的寻址方式可分为8种。深入理解寻址方式，清楚何种寻址方式适用于何种指令，对深入理解指令执行原理，正确、合理地使用指令非常重要。

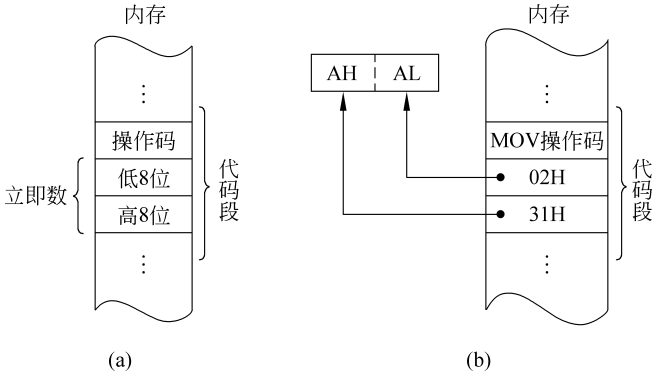
◇3.2.1 立即寻址

立即寻址(immediate addressing)方式只针对源操作数。此时源操作数是一个立即数，它作为指令的一部分，紧跟在指令的操作码之后，存放于内存的代码段中。在CPU取指令时随指令码一起取出并直接参加运算。这里的立即数可以是8位或16位的整数。若为16位，则存放时低8位存放于低地址单元(AL)，高8位存放于高地址单元(AH)，如图3-4(a)所示。立即寻址方式主要用于给寄存器或存储单元赋初值。

【例3-1】 执行指令：MOV AX,3102H。

题目解析：该指令将16位的立即数3102H送入累加器AX。指令执行后，AH=31H，AL=02H。

解：这是一条三字节指令，其执行情况如图3-4(b)所示。



◆图3-4 立即寻址方式及指令执行情况

◇3.2.2 直接寻址

直接寻址方式表示参加运算的数据存放在内存中，存放的地址由指令直接给出。即，指令中的操作数是内存操作数，[ ]内用16位常数表示存放数据的偏移地址，数据的段地址默认为数据段，可以允许段超越(segment override)。

段超越也称段重设。它通过段超越前缀(segment override prefix)修改指令操作数默认的逻辑段。段超越前缀可以附加到任何指令的内存操作数前边。

【例3-2】 执行指令：MOV AX,[3102H]。

题目解析：该指令的源操作数为直接寻址，执行结果是将数据段中偏移地址为3102H和3103H的两个单元的内容送到AX中。之所以传送两字节，是因为指令中的目标操作数是16位操作数AX。

解：假设DS=2000H，则直接寻址的操作数的物理地址为

20000H＋3102H＝23102H

指令的执行情况如图 3-5 所示。

需要特别注意直接寻址方式与立即寻址方式的区别。直接寻址指令中的常数是存放运算数据的内存单元的 16 位偏移地址，而不是数据本身。为了区分二者，指令系统规定偏移地址必须用方括号括起来。例如，在例 3-2 中，指令的执行不是将立即数 3102H 送入累加器 AX，而是将偏移地址为 3102H 的两个内存单元的内容送入 AX。

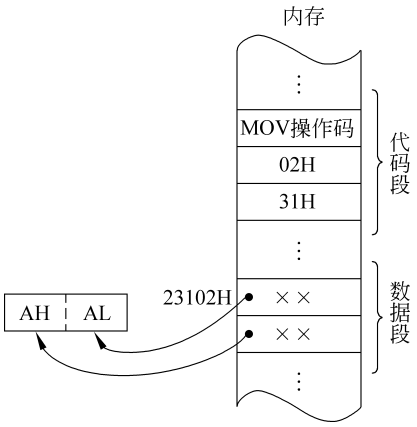
在直接寻址方式中，操作数默认在数据段[①]，但允许段重设，即可以将操作数重设到附加段甚至堆栈段中。此时在指令中要用段重设符加以声明。

例如，若将例 3-2 中的指令改为“MOV BL，ES:[3102H]”，则操作数的偏移地址虽依然为 3102H，但段地址已改变到附加段。而且由于此时目标操作数是 8 位寄存器 BL，指令的执行仅将 3102H 指向的字节单元内容送 BL。

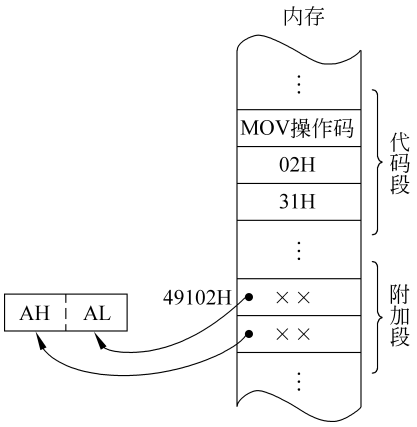
假设 ES＝4600H，则寻址的操作数的物理地址为

4600H＋3102H＝49102H

指令的执行情况如图 3-6 所示。



◆图 3-5　直接寻址方式



◆图 3-6　直接寻址方式中的段重设

由于常数无法作为地址指针，故在非机器语言的程序设计中，通常不会用常数直接表示操作数的地址，而会用字符串表示的符号地址代替。在上例中，若用 BUFFER 代替偏移地址 3102H，则指令可写成

```
MOV  BL,ES:[BUFFER]
```

或

```
MOV  BL,ES:BUFFER
```

这里的 BUFFER 即等价于高级语言中的变量。了解 C 语言或其他高级语言的读者都知

① 在 Intel 指令集中，直接寻址方式的操作数默认在数据段，但具体的编译器可能有不同的解释。例如，较常用的宏汇编（编译）程序就默认直接寻址的操作数在代码段中。

道，变量需要先声明再引用。同样，在用助记符指令编写的汇编语言（将在第 4 章介绍）中，类似 BUFFER 这样的变量也必须先定义后引用。

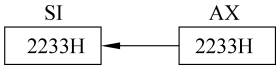
◇3.2.3 寄存器寻址

在寄存器寻址（register addressing）方式下，指令的操作数为 CPU 的内部寄存器。它们可以是数据寄存器（8 位或 16 位），也可以是地址指针、变址寄存器或段寄存器。

【例 3-3】 执行指令：MOV SI，AX。

题目解析： 该指令将 AX 的内容送到寄存器 SI 中。

解： 设指令执行前 AX＝2233H，SI＝4455H，则指令执行后 SI＝2233H，而 AX 中的内容保持不变。其执行情况如图 3-7 所示。



◆图 3-7 寄存器寻址

采用寄存器寻址方式，虽然指令操作码在代码段中，但操作数在内部寄存器中，指令执行时不必通过访问内存就可取得操作数，故执行速度较快。

◇3.2.4 寄存器间接寻址

寄存器间接寻址（register indirect addressing）是用寄存器的内容表示操作数的偏移地址。此时寄存器中的内容不再是操作数本身，而是偏移地址，操作数本身在内存中。

在寄存器间接寻址方式中，用于存放偏移地址的寄存器称为间址寄存器或地址指针。它们是 SI、DI、BX、BP。选择不同的间址寄存器，涉及的段寄存器不同。在默认情况下，选择 SI、DI、BX 作间址寄存器时，操作数在数据段，段地址由 DS 决定；选择 BP 作间址寄存器，则操作数在堆栈段，段地址由 SS 决定。但无论选择哪一个间址寄存器，都允许段重设，可在指令中用段重设符指明当前操作数在哪一个段。

简单地说，在寄存器间接寻址方式下，运算的数据存放于内存中（即出去操作数），数据在内存中的地址由间址寄存器给出。

由于间址寄存器中存放的是操作数的偏移地址，所以指令中的间址寄存器必须加上方括号，以避免与寄存器寻址指令混淆。

注意，在寄存器间接寻址方式中，存放操作数偏移地址的寄存器只允许是 SI、DI、BX 和 BP，即[ ]中不能出现其他寄存器。

【例 3-4】 已知 DS＝6000H，SI＝1200H，执行指令：MOV AX，[SI]。

题目解析： 该指令的源操作数采用寄存器间接寻址，没有设定段重设，故数据默认在数据段，段地址为 DS 的值。由已知条件可计算出操作数的物理地址为

$$60000H+1200H=61200H$$

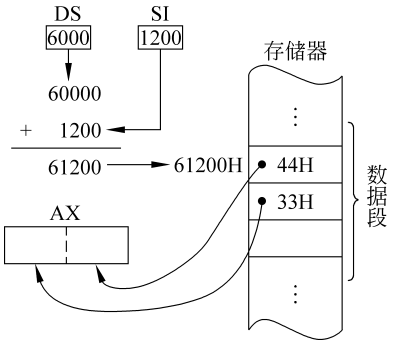
解： 指令执行情况如图 3-8 所示。执行结果为 AX＝3344H

若操作数存放在附加段，则本例中的指令应表示成以下形式：

```
MOV AX,ES:[SI]
```

【例 3-5】 若已知 SS＝8000H，BP＝0200H，执行指令：MOV BX，[BP]。

题目解析： 该指令中源操作数采用寄存器间接寻址，间址寄存器为 BP，操作数默认存



◆图 3-8 寄存器间接寻址

放在堆栈段。

解： 指令执行后，BL 为 80200H 单元中的内容，BH 为 80201H 单元中的内容。

◇3.2.5 寄存器相对寻址

在寄存器寻址方式（register relative addressing）方式中，操作数在内存中的偏移地址由间址寄存器的内容加上指令中给出的一个 8 位或 16 位的位移量得到。操作数所在的段由指令使用的间址寄存器决定（规则与寄存器间接寻址方式相同），允许段重设。寄存器相对寻址的一般格式为

指令码 目标操作数，[间址寄存器+位移量]

其中，位移量是一个常量。因为位移量可看作相对值，所以把这种带位移量的寄存器间接寻址方式称为寄存器相对寻址。

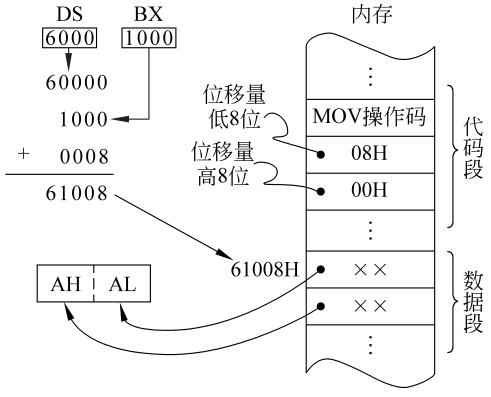
【例 3-6】 设 DS=6000H，BX=1000H，常量 DATA=8，给出指令 MOV AX，DATA [BX] 的寻址过程。

题目解析： 该指令的源操作数采用寄存器相对寻址方式，间址寄存器为 BX，故操作数默认在数据段。

解： 由题目可知，源操作数的物理地址为

$$60000H+1000H+0008H=61008H$$

指令的执行情况如图 3-9 所示。执行结果为 AX=5566H。



◆图 3-9 寄存器相对寻址

寄存器相对寻址常用于存取表格或一维数组中的元素——把表格的起始地址作为位移量，把元素的下标值放在间址寄存器中（反过来也可以），这样就可以存取表格中的任意一个元素。

【例 3-7】 设内存中有一维字符数组，其首地址（偏移地址）为 TABLE。现要读取数组中的第 10 个字符，并存放到 AL 中。

题目解析： 字符数组中的每个元素均为一字节，且第 1 个元素的下标为 0。对一维数组的寻址可以采用寄存器相对寻址方式。

解： 实现上述功能的程序段如下。

```
MOV SI,9                    ;第 10 个元素的位移量为 9
MOV AL,[TABLE+SI]           ;第 10 个元素的偏移地址为 TABLE+9
```

在汇编语言中，寄存器相对寻址指令的书写格式允许有几种不同的形式。例如，以下几种写法完全等价：

```
MOV AL,DATA[SI]
MOV AL,[SI]DATA
MOV AL,DATA+[SI]
MOV AL,[SI]+DATA
MOV AL,[DATA+SI]
MOV AL,[SI+DATA]
```

本书中的字符常量，如上式中的 DATA，均表示任意 8 位或 16 位常数。在实际指令中，通常用某一具体常数取代。例如：MOV AL,[SI+5]。

◇3.2.6 基址-变址寻址

与直接寻址类似，用位移量（某个常数）表示一维数组的首地址难以用指令实现对地址的修改，由此就引入了基址-变址寻址方式。这种寻址方式由一个基址寄存器（BX 或 BP）的内容和一个变址寄存器（SI 或 DI）的内容相加而形成操作数的偏移地址，数据所在的逻辑段由基址寄存器决定。即，默认情况下，指令中若用 BX 作基址寄存器，则数据在数据段；若用 BP 作基址寄存器，则数据在堆栈段。这两种情况均允许段重设。

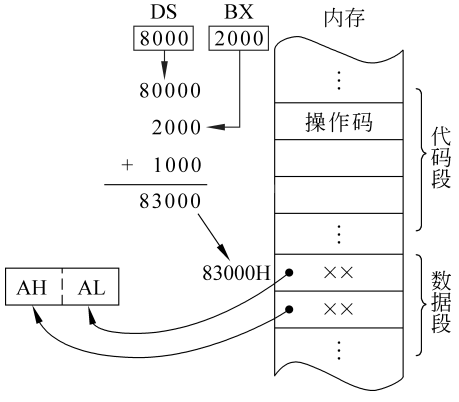
【例 3-8】 设 DS=8000H，BX=2000H，SI=1000H，给出指令 MOV AX,[BX][SI]的寻址过程。

题目解析： 该指令源操作数为基址-变址寻址，因基址寄存器使用 BX，且没有段重设，故数据默认在数据段。

解： 由题知，源操作数的物理地址为

$$80000H+2000H+1000H=83000H$$

源操作数的寻址过程如图 3-10 所示。



◆图 3-10 基址-变址寻址

指令执行后，AL=[83000H]，AH=[83001H]。

注意，使用基址-变址方式时，不允许将两个基址寄存器或两个变址寄存器组合在一起寻址，即指令中不允许同时出现两个基址寄存器或两个变址寄存器。例如，以下指令是非法的：

```
MOV AX,[BX][BP]          ;错误!同时出现两个基址寄存器
MOV AX,[SI][DI]          ;错误!同时出现两个变址寄存器
```

◇3.2.7 基址-变址-相对寻址

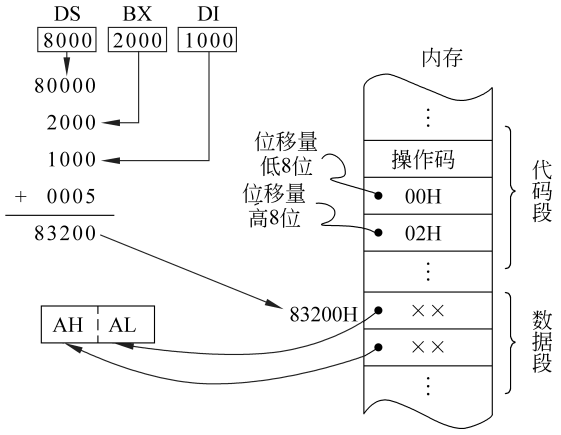
这种寻址方式是基址-变址寻址方式的扩充。指令中指定一个基址寄存器、一个变址寄存器和一个 8 位或 16 位的位移量，将三者相加就得到操作数的偏移地址。至于默认的段寄存器，仍由指令使用的基址寄存器决定。这种寻址方式允许使用段重设。

使用基址-变址-相对寻址方式可以很方便地访问二维数组。例如用位移量指定数组的首地址（偏移地址），用基址寄存器和变址寄存器分别存放数组元素的行地址和列地址，通过不断修改行、列地址，就可以直接访问二维数组中的各个元素。

【例 3-9】 若设 DS=8000H，BX=2000H，DI=1000H，给出指令 MOV AX,5[DI][BX]的寻址过程。

题目解析： 该指令中的源操作数采用基址-变址-相对寻址方式，基址寄存器为 BX，且未使用段重设，故操作数默认在数据段。该指令将段地址为 DS、偏移地址为 BX+DI+5 的连续两个存储单元的内容送入 AX。

解： 由题目中给出的条件，该指令执行情况如图 3-11 所示。



◆图 3-11 基址-变址-相对寻址

使用这种寻址方式可以很方便地访问二维数组。例如，用基址寄存器存放数组的首地址（偏移地址），而用变址寄存器和位移量分别存放行和列的值，指令就可以直接访问二维数组中指定的行和列的元素。

与寄存器间接寻址方式类似，基址-变址-相对寻址指令同样也可以表示成多种形式，例如：

```
MOV AX,DATA[SI][BX]
MOV AX,[BX+DATA][SI]
MOV AX,[BX+SI+DATA]
MOV AX,[BX]DATA[SI]
MOV AX,[BX+SI]DATA
```

同样，基址-变址-相对寻址也不允许在指令中同时出现两个基址寄存器或两个变址寄存器。即下列指令也是非法的：

```
MOV AX,DATA[SI][DI]          ;错误!同时出现两个变址寄存器
MOV AX,[BX][BP]DATA          ;错误!同时出现两个基址寄存器
```

◇3.2.8 隐含寻址

在上述各类寻址方式中，针对的都是指令中显式呈现的操作数，给出的示例都是双操作数格式指令。在 x86 的 16 位指令集中，有一部分指令为单操作数或零操作数格式。这类指令的操作数为隐含存在的，在指令的操作码中，不仅包含了操作的性质，还隐含了部分操作数的地址。这种将一个操作数隐含在指令码中或全部操作对象都隐含在操作码中的寻址方式就称为**隐含寻址**。

既然操作对象隐含在操作码中，就意味着操作对象是确定的、不可改变的。以下用乘法指令作为例子对隐含寻址进行解释。乘法指令的一般格式为

```
MUL 操作数
```

这条指令在形式上为单操作数指令。但我们都知道，乘法一定需要两个操作数：被乘数和乘数，而指令中只给出了一个操作数，显然隐含了另一个操作数。事实上，该指令显式给出的操作数是乘数的存放地址，被乘数以及乘积的地址为隐含给出的（也就是固定的地址）。有关 MUL 指令的详细解释参见 3.3.2 节，这里先用一个简单示例说明隐含寻址的概念。

【例 3-10】 说明指令 MUL BL 中操作数的寻址过程。

题目解析： *按照 MUL 指令的基本格式，操作数 BL 中存放的是乘数，指令隐含的被乘数存放在 AL 寄存器中，乘积则存放在 AX 中。*

解： 该指令的执行过程为 AL×BL→AX。该指令隐含了被乘数 AL 及乘积 AX。

3.3 Intel x86-16 指令集

x86-16 指令集按照功能可分为 6 类：

（1）数据传送指令。

（2）算术运算指令。

（3）逻辑运算和移位指令。

（4）串操作指令。

（5）程序控制指令。

（6）处理器控制指令。

严格地讲，指令是指用 0 和 1 表示的机器语言指令，但由于记忆上的困难，实际编程中都采用汇编指令而非机器语言指令。汇编指令用助记符描述，与机器语言指令一一对应。附录 C 中给出了 x86-16 指令集中的 6 类 111 条指令的助记符描述。

20 世纪 70 年代，美国加州大学伯克利分校通过对大量程序的研究，归纳出了 CISC 指令系统的计算机中存在的“80/20 规律”：20%的指令在各种应用程序中的出现频率占整个指令系统的 80%。基于此规律，本节主要对 x86-16 指令集中最常用的部分指令做详解介绍。

表 3-1 是本章常用符号说明。

表 3-1 本章常用符号说明

常 用 符 号	符 号 含 义
OPRD	泛指各种类型的操作数
mem	内存操作数
acc	累加器操作数
src	源操作数
dest	目标操作数
reg16	16 位寄存器
port	输入输出端口，可用数字或表达式表示
DATA	8 位或 16 位立即数


通用数据传送指令 01

◇3.3.1 数据传送指令

数据传送指令是程序中使用最为频繁的一类指令，无论程序功能如何，都需要将原始数据、中间运算结果、最终结果及其他信息在 CPU 的寄存器和存储器之间进行传送。数据传送指令中的绝大多数都不会对状态寄存器 FLAGS 产生影响。

按照功能划分，数据传送指令包括通用数据传送指令、目标地址传送指令、输入输出指令、转换指令和标志传送指令 5 种。以下详细介绍部分最常用的指令。

1. 一般数据传送指令 MOV

MOV 指令是几乎所有程序段中都会使用的最常用的指令之一。指令格式为

```
MOV dest,src                    ;(dest)←(src)
```

这里，dest 表示目标操作数，src 表示源操作数。该指令的功能是将数据从源地址传送到目标地址，而源地址中的数据保持不变。也就是说，MOV 指令实际上完成了一次数据的复制。

在汇编语言中，规定具有双操作数的指令必须将目标操作数写在前面，将源操作数写在后面，二者之间用一个逗号隔开。

1）指令特点

MOV 指令是最普通、最常用的传送指令，它具有如下几个特点：

(1) 该指令中的操作数可以是 8 位或 16 位的寄存器或存储单元,源操作数可以是立即数。

(2) 该指令可以使用 2.2 节讨论的各种寻址方式。

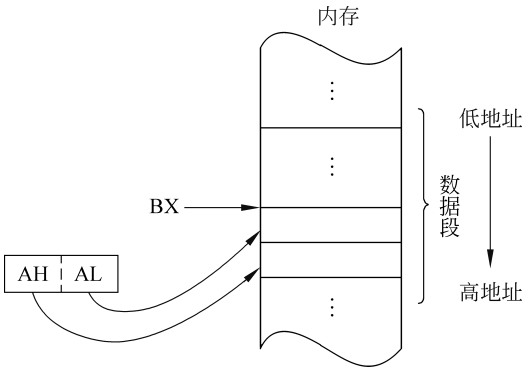
2) 指令实现的操作

MOV 指令可以实现以下各种数据传送:

(1) 寄存器与寄存器或通用寄存器与段寄存器之间的数据传送。例如:

```
MOV BX,SI                ;变址寄存器 SI 中的内容送入基址寄存器 BX
MOV DS,AX                ;累加器 AX 中的内容送入段寄存器 DS
MOV AL,CL                ;通用寄存器 CL 中的内容送 AL
```

(2) 寄存器与内存之间的数据传送。MOV 指令可以在寄存器与内存之间进行数据传送。若传送的是字操作数,则对连续两个内存单元进行存取,且寄存器的高 8 位对应内存的高地址单元,低 8 位对应内存的低地址单元。图 3-12 为指令"MOV [BX],AX"的执行原理。它将累加器 AX 的值送到间址寄存器 BX 指向的两个内存单元中。



◆图 3-12 指令"MOV [BX],AX"的执行原理

(3) 立即数到寄存器的数据传送。例如:

```
MOV AL,5                 ;将立即数 5 送入累加器 AL
MOV BX,3078H             ;将立即数 3078H 送入寄存器 BX
```

指令中立即数的字长由另一个操作数确定。

(4) 立即数到内存的数据传送。例如:

```
MOV BYTE PTR[BP+SI],5    ;将 5 送入堆栈段中偏移地址为 BP+SI 的单元中
MOV WORD PTR[BX],1005H   ;将 1005H 送入数据段中偏移地址为 BX 和 BX+1 的两个单元
```

由于指令中的立即数和内存操作数都属于字长不确定操作数。当指令的两个操作数的字长都不确定时,x86 指令系统要求必须明确指定其中一个操作数的字长,指定的方法是使用属性运算符 PTR(详见 4.1.2 节)。

(5) 内存与段寄存器之间的数据传送。例如:

```
MOV DS,[1000H]          ;将数据段中偏移地址为 1000H 的字单元内容送入数据段寄存器 DS
MOV [BX],ES             ;将附加段寄存器 ES 的内容送入数据段中 BX 指向的字单元
```

3）指令对操作数的要求

指令对操作数有以下要求：

（1）MOV 指令中两个操作数的字长必须相同，可同为字节操作数或同为字操作数。

（2）两个操作数不能同时为内存操作数。若要在两个内存单元之间进行数据传送，需要用两条 MOV 指令实现。

（3）不能用立即数直接给段寄存器赋值（要实现此功能，需使用两条 MOV 指令）。

（4）两个操作数不能同时为段寄存器。同样，要实现段寄存器到段寄存器的数据传送，需两条 MOV 指令。

（5）一般情况下，指令指针 IP 及代码段寄存器 CS 的内容不通过 MOV 指令修改，即它们不能作为目标操作数，但可以作为源操作数。

（6）虽然许多指令的执行都对状态寄存器 FLAGS 的标志位产生影响，但通常情况下，FLAGS 整体不作为操作数。

实际编写程序时，有时需要将内存一个区域中若干单元的数据（称为数据块）传送到另外一个区域，或是向若干单元赋同样的值（比如清零）。对于这种重复性的工作，计算机是最容易的。下面就通过一个例子说明如何利用 MOV 指令完成数据块的传送。

【例 3-11】 将内存数据段偏移地址从 1000 起始的 200 字节送入首地址为 2000 的区域中。

题目解析： 由于 MOV 指令不支持内存单元间的直接数据传送，因此需要用两条 MOV 指令实现。在这里，我们当然不希望用 400 条 MOV 指令完成这 200 个单元数据的传送。较好的实现方式是通过循环程序实现这个数据块的传送。请看下面的程序段。该程序段中某些指令还没有介绍，这里先使用它们。

```
        MOV SI,1000         ;源数据块首地址（偏移地址）送入 SI
        MOV DI,2000         ;目标首地址（偏移地址）送入 DI
        MOV CX,200          ;数据块长度送入 CX，即循环次数为 CX
NEXT:   MOV AL,[SI]         ;源数据块中第一个字节送入 AL
        MOV [DI],AL         ;AL 内容送入目标地址，完成一字节数据的传送
        INC SI              ;SI 加 1，修改源地址指针
        INC DI              ;DI 加 1，修改目标地址指针
        DEC CX              ;CX 减 1，修改循环次数
        JNZ NEXT            ;若循环次数（CX）不为 0，则转移到 NEXT 标号处
        HLT                 ;停止
```

通用数据传送指令 02

2. 堆栈操作指令 PUSH 和 POP

堆栈操作指令针对内存堆栈段进行操作，其段地址放在堆栈段寄存器 SS 中。

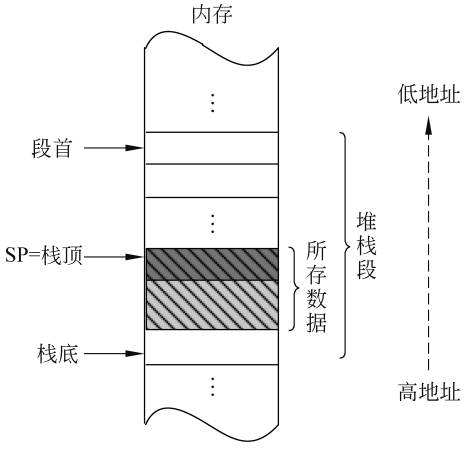
堆栈段是内存中一个特定的区域[①]，用于存放寄存器或内存中暂时不用又必须保存的

① 这里的堆栈特指 1.1.5 节中提到的栈，是从内存管理的角度而非数据结构的角度描述的在动态分配内存时指定的一类存储区域。

数据。

与数据段等其他逻辑段更关注段地址不同，堆栈段更注重**栈顶**和**栈底**，如图 3-13 所示。栈顶(偏移)地址由堆栈指针 SP 给出，表示目前堆栈段中已存储数据的容量。栈底是指堆栈段中的最高地址单元；栈底到栈顶之间为已存储的数据；而栈顶到段首之间是预留的存储空间，可以继续存放数据。所以：

- 若栈顶＝栈底，表示空栈。
- 若栈顶＝段首，表示满栈。



◆图 3-13 堆栈段

对堆栈段的操作必须遵循以下原则：

(1) 对堆栈段的存取必须以字为单位。在 16 位指令集中，堆栈指令中的操作数必须是 16 位。

(2) 向堆栈段中存放数据时，总是从高地址向低地址方向增长；而从堆栈段中取数据时方向正好相反。

(3) 堆栈段在内存中的位置由 SS 决定，堆栈指针 SP 总是指向栈顶，即 SP 的内容等于当前栈顶的偏移地址。所谓栈顶，是指当前可用堆栈操作指令进行数据交换的存储单元。在压入操作数之前，SP 先减 2；每弹出一个字，SP 加 2。

(4) 对堆栈段的操作遵循后进先出的原则。

在程序中，堆栈段主要用于子程序调用、中断响应等操作时的参数保护，也可用于实现参数传递。

1) 压栈操作指令 PUSH

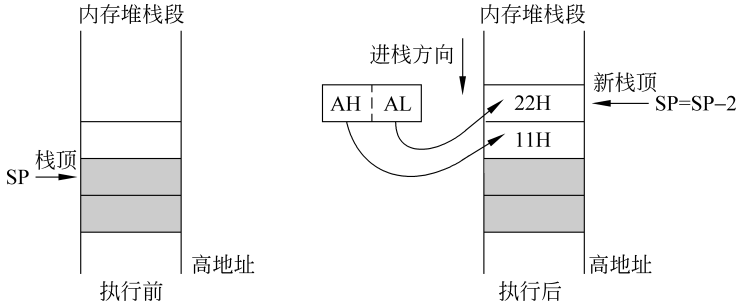
PUSH 指令将指令指定的字操作数压入堆栈。指令格式为

```
PUSH  src
```

这里，src 是源操作数，隐含的目标操作数为栈顶地址的内容。PUSH 指令的执行过程如下：

```
SP-2 →SP
src 高 8 位 → [SP+1]
src 低 8 位 → [SP]
```

图 3-14 给出了 PUSH AX 指令执行前后堆栈段的变化情况。这里假设 AX＝1122H。由图 3-14 可见，PUSH 指令是将 16 位的源操作数送到堆栈的顶部。



◆图 3-14　PUSH AX 指令执行前后堆栈段的变化情况

2）出栈操作指令 POP

POP 指令将当前栈顶的一个字复制到指定的目标地址，并紧接着使堆栈指针 SP＋2，指向新的栈顶位置。指令格式为

```
POP dest
```

这里的 dest 是目标操作数，指令隐含的源操作数是栈顶地址的内容。POP 指令的执行过程如下：

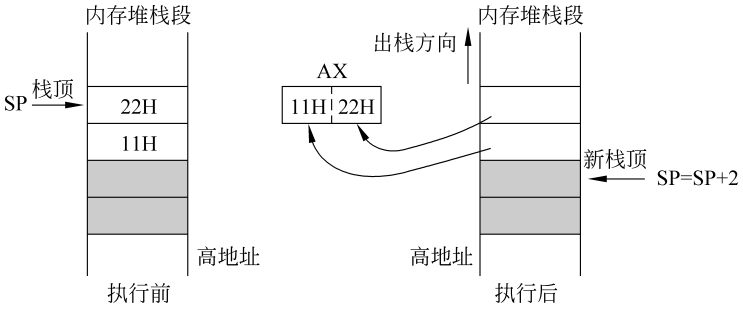
```
[SP] → dest 低 8 位
[SP+1] → dest 高 8 位
SP+2 →SP
```

堆栈操作指令在使用时有两点需要注意：

（1）PUSH 和 POP 指令的操作数都必须是字操作数，它们可以来自 16 位的通用寄存器或除 CS 之外的段寄存器（虽然 PUSH CS 指令合法，但 POP CS 指令非法），也可以来自内存（地址连续的两个存储单元），但不能是立即数。

（2）若操作数来自内存，则需要用属性运算符（PTR）说明其字长。PTR 的功能是说明其后的操作数的字长是其前边字符串表述的含义（详见第 4 章）。

图 3-15 给出了 POP AX 指令执行前后堆栈段的变化情况。这里依然设 AX＝1122H。



◆图 3-15　POP AX 指令执行前后堆栈段的变化情况

例如：

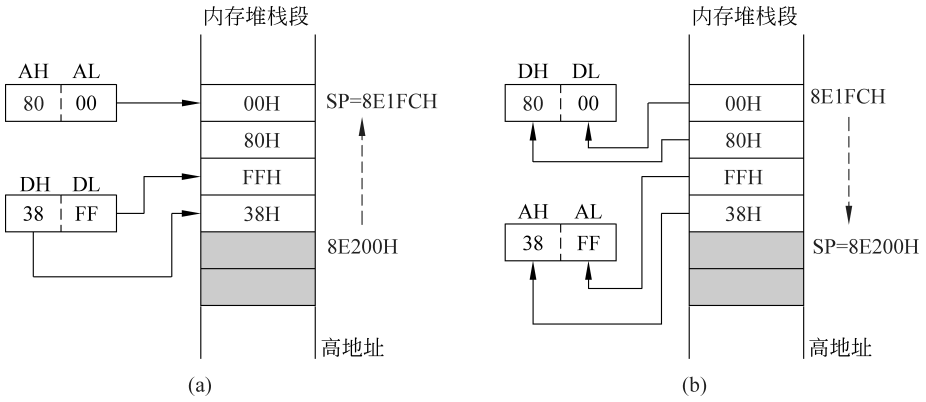
```
PUSH AX                    ;通用寄存器内容压入堆栈
PUSH WORD PTR[56+SI]       ;数据段中两个连续存储单元内容压入堆栈
POP DS                     ;从栈顶弹出一个字到段寄存器中
POP WORD PTR[BX]           ;从栈顶弹出一个字到数据段的两个连续存储单元中
```

在程序中，PUSH 和 POP 指令一般成对出现，且执行顺序相反，以保持堆栈原有状态。当然，在必要时也可通过修改 SP 的值恢复堆栈原有状态。

【例 3-12】 用图表示如下程序段的执行过程。

```
MOV AX,8000H
MOV SS,AX
MOV SP,0E200H
MOV DX,38FFH
PUSH DX
PUSH AX
⋮
POP DX
POP AX
```

题目解析：该程序的前 3 行语句设置堆栈段的段地址为 8000H，栈顶指针为 E200H[①]。执行两条 PUSH 指令后，寄存器 DX 和 AX 的值被分别压入堆栈保存，如图 3-16(a)所示。在执行完相关代码后(程序中省略的部分)，再执行两条 POP 指令，分别将栈顶地址中的内容弹出到 DX 和 AX 中，如图 3-16(b)所示。



◆图 3-16 按先进先出原则执行的堆栈操作

由图 3-15 可见，执行后 AX 和 DX 的内容实现了互换，并没有实现对原寄存器中内容的保存(保存数据是堆栈的主要作用)。形成这样的结果的主要原因是没有按照后进先出的堆栈操作原则，而是按先进先出原则进行了堆栈操作，从而实现了 AX 和 DX 内容的互换(有时可利用堆栈的这一特点，实现两个操作数内容的互换)。

① 汇编语言规定：如果作为操作数的立即数的首位是字母(如 E200H)，则该操作数前边要加 0，以便编译器识别。