嵌入式系统原理与实验（上）

[键入文档副标题]

上海交通大学嵌入式系统原理与实验课程组

2011/2/1

目录

第1章.　计算机基础

电子数字计算机是一种能够自动、高速、精确地进行信息处理的现代化的数字电子系统，其基本特点是运算速度快、信息存储容量大、具有逻辑判断能力。

电子计算机系统分为硬件和软件两大部分，硬件是由物理元件（电子元件）构成的数字电路系统；软件是由程序代码（二进制代码）构成的数字系列。硬件与软件一起构成完整的计算机系统，本章将介绍计算机各组成部件的构成，介绍软件和硬件的概念。

1.1. 计算机的基本构成

计算机的基本功能是存储和处理各类信息，并将处理的结果向外界输出。为了完成这些基本功能，要求计算机能够自动地输入信息、处理信息、存储信息以及输出信息，计算机的基本部件就是根据这些要求设置的，分别用一个部件完成上述一个功能，所以计算机的主要部件有输入单元、运算单元、存储单元、输出单元组成，再加上用一个控制器单元来控制各种功能部件协同工作。

在计算机的发展初期，美籍匈牙利科学家约翰·冯·诺依曼（ John von Neumann，1903－1957）提出了计算机的基本模型，计算机由运算器、存储器、控制器、输入单元和输出单元这五个基本组成部分，世人称之为约翰·冯·诺依曼结构。计算机通过输入单元输入数据和程序，经过运算器后，存放在存储器中，在运算时将存储器中的程序送给控制器，控制器根据程序对运算进行控制，将数据送到运算器，运算的结果先存储在存储器中,在适当的时候通过输出单元输出。这种结构以运算器为中心，数据的输入输出都经过运算器进行，在存储器和运算器之间形成一个数据流，在存储器与控制器之间形成一个指令流，从控制器到其他各组成部分还有控制信息流。计算机的基本结构框图如图 1-1 所示，图中标出了各组成部分之间的数据和控制信息的流动关系。尽管经过几十年的发展，计算机在许多方面都发生了巨大的变化，但仍然可以划分为上述 5 个基本部件，指令和数据的流向也基本未变。当然，不同的计算机有不同的部件数量，部件之间有不同的连接结构，部件的内部结构也各个不相同。

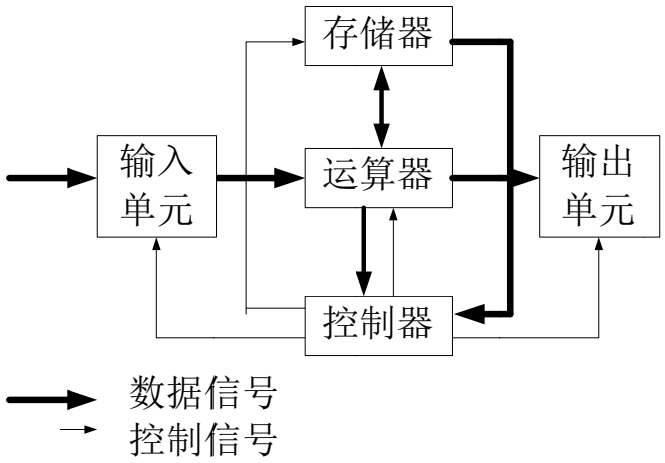


图 1-1 计算机的基本结构

下面首先介绍这些硬件的部件及其相互连接。

1.1.1. 运算器

计算机的一个特点就是具有很强的运算能力，运算器是完成运算功能的部件、运算器中主要包含算术逻辑单元（ALU，Arithmetic Logic Unit），简称算术逻辑单元。算术逻辑单元是一个实现数学运算的数字电路，执行如加、减、乘、除等各种数值运算和与或非等逻辑运算和操作。计算机中采用二进制的数据表示方式，计算机中的数据是外部信息在计算机中的编码表示形式。运算器的运算功能是对二进制的数据进行的运算。

算术逻辑单元有两个数据信号输入端和一个数据信号输出端，可同时输入两个参加运算的操作数，运算完成后输出一个结果数据。算术逻辑单元是一个组合逻辑的数字电路，不具有记忆功能，不能保存数据，电路的输出端的数据取决于当时的输入端的数据。输入端的数据消失后，输出端的运算结果也就随之消失。在进行算术和逻辑运算操作时，需要先将数据从存储器中取出，放到运算器中，在计算完毕后再存放到存储器中。为了保存运算中所需要的数据，在运算器中有若干个临时存放数据的寄存器，用于存储计算过程中最频繁使用的数据，如一些中间运算结果等。除了存放运算结果，寄存器中还可保存运算的状态，如数据是否有进位、是否为零、是否溢出、是否发生了其他错误以及错误的原因等，以便对运算中出现的各种情况进行处理。在计算机中有多种不同的寄存器，在指定一个寄存器时可以给每个寄存器指定一个编号，称为寄存器号。寄存器号在电路中可用二进制代码识别。

1.1.2. 存储器

存储器的作用是存储程序和数据。存储器中采用某种存储介质存放数据和程序代码信息，主要的存储介质有半导体电路、磁性记录介质、光存储介质等。存储器中可包含的信息数量称为存储器的容量。

半导体介质的存储密度高，工作速度快，在电源切断时存储的内容会丢失；而光、磁等介质则容量大，但工作速度低。实现存储器的元器件有半导体存储器芯片、磁盘、磁带、光盘等。为了用最合理的 成本实现最大的存储容量，存储器都分成主存储器（Primary Storage 或 Memory）和辅助存储器（Secondary Storage）。主存储器又称主存或内存，它一般采用半导体存储器件实现，速度较高，程序和数据在运行时主要放在主存中，由于半导体存储器的成本较高，难以实现很大容量的存储器。为此需要附加一个成本较低、容量更大的辅助存储器。辅助存储器用于存放一些在计算过程中不频繁使用的数据和程序。辅助存储器称外存，因为它们一般是通过输入/输出单元连接到主存储器的。计算机处理的信息必须具有某种适当的表示形式。在数字计算机中，信息以二进制代码的形式表示，二进制代码的运算是构成数字系统的基础。

在数字计算机中，任何数字、文本中的字符或者指令都表示成二进制的编码数据。其中每个二进制数据代码称为“位”（Bit，b），它是数据的最小表示单位。信息通常表示为若干个位，通过这种位的组合，计算机中就可以表示各种数据和字符，如十进制数值和英文字母。这种把数据信息表示成二进制位组合的表示过程称为编码，每个二进制位组合又称为代码，它是编码的结果。一个二进制位可以有 0 和 1 两个代码，2 个二进制的位可以有 4 个代码（00，01，10，11），3 个二进制的位可以有 8 个代码（000, 001，010，011，100，101，110，111），10 个二进制位可构成 1024 个代码。

主存储器由大量的数据存储单元构成，每个存储单元可存储一位信息。数据的存储操作一般是以“字”（Word）为单位进行。字是运算器中进行的数据运算的单位。衡量运算数据

的基本单位是字，它是运算器一次可以完成的运算位数。一个字的数据运算操作在一次运算操作中完成，大于一个字的数据必须分步完成，因此计算机的字长反映了计算机中并行计算的能力。对不同的计算机，字包含的位数可能是不同的。有的计算机中一个字是 16 位的，有的计算机中一个字是 32 位的。一个数据字中包含的位数称为该计算机的字长。通常，计算机的字长分为 8 位、16 位、32 位和 64 位等。在一些较早的计算机中，由于历史原因，把 16 位数据称为一个字，尽管它实际上能够进行 32 位或 64 位的运算。这种计算中，把 32 位的数据称为一个双字，把 64 位的数据称为一个 4 倍字。

在计算机中，一个字节是由 8 位构成的编码单位。它是衡量存储器容量的基本单位。存储器容量的单位为字节数（Byte 或 B）、千字节数（KB）、兆字节数（MB）以及吉字节数（GB）等。其中 1 个字节等于 8 位，若干个字节构成一个字。存储器容量单位的换算关系是：1KB=1024B，1MB=1024KB，1GB=1024 MB，1TB=1024GB。通常，我们用小写 b 表示位，用大写 B 表示字节。例如，8b 表示 8 位，8B 表示 8 字节，8W 表示 8 个字。

为了寻找主存储器中的某一个存储单元的位置，需要给不同的存储位置指定一个编号。这个编号就是存储器的地址器（Address）。主存储器的地址是一个依次编排的数字，也就是一串序号。地址是识别存储器中不同存储单元的唯一标志，不同的存储单元有不同的地址。对存储器中存储位置进行数据写入和读出通过指定一个地址进行。存储器中的存储位置的指定通常是以字节为单位，即给每个字节的存储位置指定一个地址。

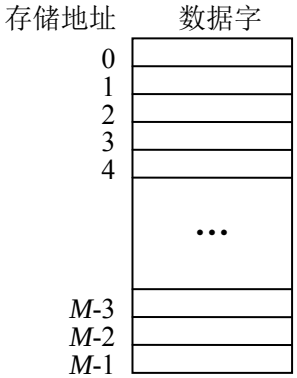


图 1-2 存储器

对于一个能存储 M 个字节的存储器，其地址通常编制为从 0 到 M-1 的数表示存储器中连续的存储位置（见图 1-2）。存储器地址的长度决定了能够编址的存储位置的数量。如果用 8 位代码表示存储器地址，那么只能对 256 个存储位置进行编址。现在大多数计算机采用 32 位地址，这样它能够配置的存储器容量为 4GB（2^{32} 字节）。

对于存储器的操作有两种，读操作和写操作。存储器的读操作是将存储器中的内容传送到运算部件中，而存储器中的内容不变。在进行读操作时，运算部件必须向存储器送一个数据地址以及读操作信号。存储器的写操作是将运算部件中的数据传送到存储器中的某个存储位置存储起来，该存储位置中原来的数据被擦除。在进行写操作时，运算部件必须向存储器送数据地址、数据以及写操作信号。存储器的数据读操作和写操作统称为存储器的访问（Access）。在访问存储器中的数据时，我们需要向存储器提供数据的地址。由于存储器的地址是以字节为单位的编号，所以对存储器的访问也是以字节为最基本的单位。每次访问存储器时至少读写一个字节的数据，可以连续读写多个字节。在计算机的工作过程中不断地进行存储器的访问，存储器访问从地址的输送开始，以数据的输送为结束，完成的各个步骤形

成一个访存周期。访存周期所经历的时间称为存储器周期时间，它是一次访问存储器中的数据所需的时间。

我们可以把存储器看成是一个由大量寄存器构成的寄存器组，寄存器与存储器的一个重要区别是：对寄存器的访问采用寄存器号，对存储器的访问则根据地址进行。从硬件角度，寄存器号和存储器地址都是一个二进制的代码。但是寄存器数量要比存储器的字数少得多。所以寄存器号比存储器地址短得多。一般寄存器号在 4 到 5 位左右；而存储器地址则一般在 32--64 位之间。寄存器与存储器的另一个区别是访问的速度，寄存器在内部，距离运算单元很近，访问的速度很高；而存储器由于复杂得多，而且距离的距离相对较远，所以访问的速度较慢。为了减少数据在运算器与存储器之间频繁传递造成的时间延迟，许多计算机中设置了较多的寄存器。运算数据先从存储器中读到寄存器中，然后进行运算，运算的中间结果也尽量放在寄存器中，最后结果写回存储器。

存储器的访问在控制器的控制下进行。因为程序和数据可存放在存储器中的任何位置，因此存储器访问时应当能够快速方便地访问任何地址中的内容，访问的速度应与存储位置无关，与访问的方式（读或写）无关。具有这种功能的存储器称为随机访问存储器（Random Access Memory）或称 RAM。计算机的主存储器主要用实现。实现存储器的元器件有静态存储器（SRAM，Static RAM）芯片和动态存储器（DRAM，Dynamic RAM）芯片两种类型。

除了随机访问存储器外，还有按顺序访问的存储器或者部分顺序访问的存储器，如磁盘和磁带等。对这些存储器的访问时间与数据在存储器中的位置有关。顺序访问存储 器是完全串行访问的存储器，如磁带，信息以串行的方式从存储介质的始端开始写入。部分顺序存储器是部分串行访问的存储器，如磁盘和光盘，它介于顺序访问和随机访问之间。对信息的访问包括两个操作：将访问的起始位置移动到任意一个指定的区域，接着对这个区域中的数据按顺序访问，前一个操作是随机方式的，后一个操作是顺序方式的。

1.1.3. 控制器

控制器控制和协调其他单元的工作，它在计算机指令的控制下进行工作。计算机指令是一种经过编码的操作命令，它指定需要进行的算术和逻辑操作，指定计算机中信息的传递以及在计算机与输入/输出设备之间的信息传递。指令在计算机用二进制的代码表示，以便于硬件的识别。不同的计算机一般有不同的指令。控制器对指令进行译码，并根据指令生成一系列时序控制信号，控制其他单元的工作，使得各个部件协作完成某一件事情。例如，控制运算器完成一个加法操作、控制某一个寄存器把数据送到运算器、控制某一个寄存器把数据送到存储器、控制加法器接收从寄存器送来的数据等。它根据计算机指令进行控制工作。

一条计算机指令的功能是有限的，完成复杂的运算功能需要将多条指令组合起来， 构成一个指令序列，这样的一个完成某种功能的指令序列称为程序，一个程序能够完成较复杂的功能。在计算机中，把需要完成的复杂功能分解成一条条指令，每条指令表示一个简单的功能，许多条指令的功能组合起来构成了计算机实现的复杂功能，所以，要完成计算机实现的功能，就要完成每个指令的功能。指令的功能由计算机的硬件来完成。怎样将许多条指令组合起来完成一件复杂的工作则是程序设计人员的任务，也是软件实现的任务。将指令组合起来完成复杂功能的过程就是程序设计，也就是软件的开发。由于计算机需要完成各种不同的复杂的计算任务，不可能将所有的复杂任务都用指令来实现，所以在计算机设计中一般用指令规定比较简单的基本功能，然后由软件组合这些指令来完成各种不同的复杂功能。

程序在执行之前存储在主存储器中，控制器通常按指令存储的顺序依次执行，或者根据指令决定执行的顺序。控制器从存储器中读取每一条指令，对指令进行译码分析，并根据指令要求的功能生成一系列时序控制信号，控制其他单元的工作。在完成一条指令之后，控制器又从存储器中读取下一条指令，周而复始。控制器通常按指令存储的顺 序自动地从存储器中取出指令，并依次执行，或者根据指令决定执行的顺序。计算机就是一种在“存储程序”的控制下运行的数字运算设备。数据是编码形式的各种信息，它在计算机中通常作为程序的操作对象。在计算机中，数据可以是整数、浮点数的编码， 也可以是声音信息、图像信息的编码，还可以是程序代码等。

在计算机中，数据和指令都以二进制的形式存储。从存储的数据本身看不出它是哪一类数据，也不能分辨它是数据还是指令。控制器在取指令时，把从存储器中读出的信息作为指令处理；在取数据时把从存储器中读出的信息作为数据处理，在程序设计中需要注意。

运算器和控制器一起构成了计算机的中央处理器（CPU，Central Processing Unit），简称处理器。目前，还存在把 CPU 和小容量存储器合在一起称微控制器（MCU，Micro Control Unit ）或单片机。MCU 芯片中一般还包含输入、输出接口电路部件，MCU 面向以控制为主的应用。

1.1.4. 输入输出设备

计算机从输人设备获得外部的信息。输入设备将外部信息以一定的数据格式送入主机。输入设备如键盘和鼠标器等，键盘采集操作员的按键操作信息并将这种信息转换成数据编码。鼠标器将位置信息以数字形式输入到计算机中。其他输入设备还有书写笔和图像扫描仪等。

输出设备的功能是将计算机的处理结果提供给外部，输出设备如显示器、打印机、绘图仪等，某些设备同时兼有输入和输出的功能，如触摸屏等。

计算机的输入、输出设备通常又称为外围设备。因为它们一般包含一些机械部件等难以与主机集成的部件，或者需要卸除、移动的部件，所以通常与主机分离。外围设备是计算机不可缺少的组成部分，它是人类与计算机、计算机与计算机、计算机与其他设备交换信息的界面。人类需要通过输入设备将运算的数据以及操作要求告诉计算机；计算机需要通过输出设备将运算的结果提供给人类。计算机的输入输出设备与计算机的应用密切相关，随着计算机应用领域的不断拓展，计算机输入输出设备的类型也不断丰富和发展，近年来计算机的输入输出设备向着多媒体的方向发展，使得计算机的应用领域开辟了一个新的天地，而随着计算机应用领域的不断扩展，对输入输出设备的类型也不断提出新的要求，使得新型输入输出设备不断涌现。

外围设备通常通过输入输出接口与主机连接，磁带、磁盘和光盘等辅助存储器通常也称为外围设备，因为它们也需要通过输入输出接口与主机连接。输入输出接口是指主机与外围设备之间传递数据与控制信息的电路。一台计算机可以与多种不同的外围设备连接，因此需要有多种不同的输入输出接口。

1.1.5. 系统连接

将 CPU、存储器和输入输出设备集成在一起就构成了一个完整的计算机系统。计算机中的各功能单元之间可以有不同的连接方式。图 1-3 中给出了计算机系统五大模块之间的数据信息的流向和控制信息的流向。根据这些信息的流向可以设计出不同的计算机结构。我们

可以根据这个信息的流向，给每一个信息流向安排一条信号线路，形成与信息流向一致的计算机连接结构。比较简单的方法是通过一条或者几条称为总线的公共线路进行连接。总线是连接各个功能部件的纽带，它是一组信号线路，可同时连接计算机中多个部件，供多个功能部件共享使用。标准的总线是一束不同功能信号线的集合，包括机械尺寸和电气的规范。不同结构的计算机采用不同的总线连接方式，不同的连接方式导致不同的功能特征、性能特征和实现成本。

最为简单的计算机结构是用一条总线连接所有模块，形成总线结构的计算机硬件系统。如图 1-3 所示，图中的总线是一条 32 位数据线和 32 位地址线的总线，输入设备和输出设备与存储器一样，直接连接在总线上。总线上通过控制信号表示对存储器或者输入/输出设备的访问，控制信号 $R/\overline{W}$ 表示数据传递的方向。在这种连接方式中，CPU 对于输入输出设备的控制信息以及输入输出设备的工作状况信息是通过数据线路传递的，从而减少了连接线路的数量。在下面逻辑图中，将总线用较粗的线表示，单个信号线则用细线表示。

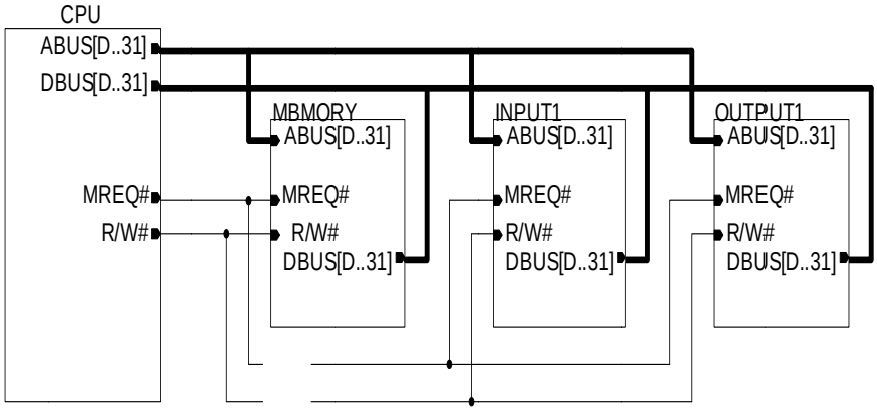


图 1- 3 计算机硬件连接结构的例子

一个计算机系统可以只包含一个 CPU，构成单处理机系统。随着微电子技术的发展，的性能迅速提高，使得单处理机系统的性能不断提高。但是许多应用场合要求提供更高的性能，为此可以在一个计算机系统中集成多个处理器以及存储器，构成并行计算机系统。随着双核和多核 CPU 芯片的出现，并行计算机系统正在迅速普及。

1.2. 计算机软件概述

计算机软件将各种计算机的指令有序地组合起来，使计算机硬件部分能正常工作并扩展其功能，完成复杂的计算任务。本节介绍计算机软件的基本概念以及与硬件的关系。

1.2.1. 软件的分类

计算机软件一般可分为系统软件和应用软件两类。系统软件是整个计算机系统的一部分，使得计算机系统的功能完整，并为应用提供一个平台。系统软件负责命令解释、运行管理、系统维护、网络通信、软件开发和输入输出管理，如操作系统、诊断程序、编译和解释程序、网络通信程序等。应用软件是面向用户应用的功能软件，专门为解决某个应用领域中的具体任务而开发。如音频视频处理等多媒体软件、印刷排版软件、电子设计自动化（EDA）软件、数据处理软件、自动控制软件等。随着计算机应用的不断发展，应用软件也不断地丰富

和完善。

应用软件、系统软件和硬件构成了计算机系统的三个层次。应用软件为用户提供一个应用系统的界面，使用户能够方便地使用计算机解决具体问题。系统软件则向用户提供一个基本的操作界面，并向应用软件提供功能上的支持。硬件系统是整个计算机系统的基础和核心，所有的功能最终由硬件完成。计算机系统按功能可划分成多层次结构，在硬件之上有操作系统级、汇编语言级、高级语言级和应用语言级；计算机硬件也是一个层次化的结构，可有微系统结构级、寄存器级与电路级等。

在实际硬件以上的所有机器层次都称为虚拟机（Virtural Machine），它们都是由软件构成的外部特性。由于计算机系统是一个非常复杂的数字系统，就像硬件的层次化结构一样，在计算机设计中我们一般将软件也分成几个层次，从而使得从某一较高层次上观察计算机时看不到较低层次的细节，这样才能使我们比较方便地了解计算机某一方面的特点。这种分层就形成了计算机系统的不同虚拟机。处在某一级虚拟机层次的程序员只需知道这一级的虚拟机特性，其下层的特性不需要知道，即下层特性对该层程序员是透明的。

计算机硬件的特征对操作系统用户和应用程序用户是透明的，就是说这些用户看不到计算机的硬件特征，这些用户可以不关心计算机系统硬件的特征。例如在同样安装了某个操作系统（如 Windows XP）之后，用户不必关心是什么厂家的，有什么特点等，因为操作都是一样的。操作系统虚拟机对应用程序用户是透明的，应用程序使得 用户可以不关心操作系统虚拟机的特征。例如在安装了某种应用软件（如网页浏览器以及数据库软件）之后，不管它运行在什么操作系统上，用户的操作都是一样的，所以用户可以不关心操作系统的特征。

计算机系统的大部分功能既可以用硬件实现，又可以用软件实现。如 64 位数据运算、浮点数据运算、图形处理等功能在某些计算机中用硬件实现，在另一些计算机中则用软件实现。计算机主机功能的这两种实现在逻辑上是等效的，其区别在于速度、成本、可靠性、存储容量、变更周期等因素。一般而言，用硬件实现的功能性能较高，同时成本也相对较高，而且硬件不易改变，它的灵活性较差。

具有相同功能的计算机系统，它们的软、硬件之间的功能分配，可在很宽的范围内变化，没有固定界线。从功能上看，软件是硬件的扩充。软件和硬件之间的界面是计算机的指令系统，即计算机全部指令的集合。影响软硬件功能划分的主要因素有性能、成本、对存储器容量的需求量、 可扩展性等。随着集成电路（IC）技术的不断发展以及器件的功能越来越强，硬件实现的功能在逐步增加。从产品形态上说，软件一般存放在磁盘或光盘上。这时，磁盘或光盘是软件的载体，而不是软件本身。

通常可把固定不变的常用软件固化在硬件中，如写入只读存储器（ROM）中，称为固件（Firmware）。固件是介于硬件和软件之间的实体。其设计方法类似于软件，而实现形态上则属于硬件。固件的例子如计算机的启动软件，包含主机最基本的驱动程序（BIOS，Basic Input Output System）。

由于不同的计算机有不同的指令集，所以在两台具有不同指令集的计算机中，一台计算机的程序不能放到另一台计算机上运行。计算机的软件兼容指软件的通用性。如果一个计算机系统上的软件能在另一个计算机系统上运行，并且得到相同的结果，则称这两个计算机系统是软件兼容的。软件的兼容需要硬件的支持。通常，计算机系统为了软件兼容设计成一个系列，在系列中能够向下兼容，即新的系统能够运行旧系统上的软件。

1.2.2. 操作系统的概念

操作系统是一个最主要的系统软件，它管理系统资源、为应用程序提供运行环境并且为用户提供操作界面。操作系统在启动计算机时开始运行，一直运行到关机。操作系统构成了一个虚拟机，它能够接受用户的操作命令，控制硬件的操作。一般而言，操作系统有以下主要功能：

（1）存储管理。存储管理分为内存管理和外存管理。内存管理主要是管理内存的分配，外存管理一般是管理磁盘存储区和文件结构。内存管理是在内存地址空间内管理各个系统程序和用户程序对于存储器的分配使用。外存媒介主要是磁盘、磁带和光盘等。 这些存储媒介的特点是成本低，但数据访问的启动速度慢，因此在访问时一般是一次访问一批数据。数据在外存中一般以文件的形式组织和管理。文件是信息集合的逻辑单位，其中存储的信息如用计算机语言编写的程序、可执行程序、数据等。一些新型操作 统还能够对有大量磁盘构成的存储系统进行管理，提高数据存储容量和可靠性。

（2）命令处理。用户给操作系统的命令通常是启动一个程序的运行，从而完成某一项系统操作或者应用操作。操作系统的命令处理功能是接收和处理用户的操作命令。接收用户命令就是分析用户操作命令的语义，处理用户操作通常是启动某一个程序的运行。命令处理功能的目的是提供一个交互式的人机界面，为用户使用计算机提供方便。传统的操作命令输入方式是命令行形式的，现在计算机的操作系统的用户操作界面普遍采用窗口菜单方式、语音和手写等多媒体方式，使用户操作起来更加方便。

（3）进程管理。进程是程序运行管理的基本单位。它包括程序代码、数据空间和控制信息。当一个程序在计算机中启动运行时，就在计算机中形成一个进程，或称任务。 计算机中可以同时启动多个程序，形成多个进程。这些进程都处于运行状态，但在某一时刻只有一个进程真正被 CPU 所处理。确定 CPU 运行哪个进程就是操作系统的进程管理功能。操作系统的进程管理是分配调度进程在 CPU 上的运行，这一功能使计算机能够同时接受和处理多个用户提交的任务，使计算机发挥更大的效用。在 Windows XP 等新型操作系统中，还引入了线程的概念。线程是进程中的一个对象，共享使用进程的程序代码。线程中包含一个线程标识符（ID）、一个线程状态（包括当前指令的地址）、一个用户态堆栈和一个系统态堆栈。

（4）设备管理。管理磁盘、鼠标器、打印机、光盘驱动器等输入/输出设备，为各种型号的外围设备提供统一的程序设计界面，并为这些设备的共享使用和管理提供方便。

（5）网络通信管理。实现某种网络的通信协议，管理计算机之间通信联系的方式，为计算机之间的操作和程序设计提供方便的界面。

此外，操作系统还提供一个公共子程序库，完成基本的操作，特别是控制硬件的操作，如控制外围设备的输入输出的操作。操作系统提供的库是一种动态连接的库，应用程序可以调用这个库中的程序。应用程序对于操作系统库函数的调用称为系统调用。操作系统为系统的操作和应用程序运行提供的界面又称为平台。计算机中把 CPU、主存储器和辅助存储器、外围设备等部件统称为资源。操作系统的各种管理功能统称为资源管理。

新型操作系统还提供病毒防护、数据加密等安全性能，以满足人们对安全性的要求。一些操作系统还提供了简单的应用程序，如游戏、计算器、音频和视频播放等。

操作系统有多种不同的类型。最早的计算机是单道程序的（单任务），在计算机中只能

有一个应用程序运行。一个程序运行完成后才能运行下一个程序，这样计算机的功能受到了局限，性能也不能得到充分发挥。为了提高计算机系统内各种资源的使用率，后来计算机中采用了多道程序技术，使得计算机中可同时运行多个程序，程序之间可进行软硬件资源共享（多任务）。每个程序在运行时以一个进程的形式存在于系统中，作为运行调度的一个单位，它包括了程序运行的状态和占用的系统资源。程序是一个静态的指令序列，进程则是程序的动态执行过程以及该程序执行时占用的系统资源。进程代表了一个资源占有的单位和一个运算的单位，它是多用户操作系统组织运算任务的手段。多任务操作系统为每个进程分配一部分系统资源，使得每个进程能够得到运行。

我们常用的操作系统是一种交互式的操作系统。这种操作系统是一个应用程序的运行平台，为应用程序的运行提供基础设施和环境。应用程序的开发是相互独立的，应用程序的运行也是独立的。应用程序可以随时加载到计算机系统中，并作为一个进程运行。应用程序运行完成后退出系统，释放所有的资源。系统中运行的各个进程的地址空间是相互分离的。在桌上型和服务器型计算机系统中，操作系统又分为内核和外围两个层次。操作系统内核提供最基础的机制，操作系统外围提供与应用程序的接口。访问操作系统的地址空间必须在 CPU 内核态模式下进行，用户进程在用户态中，它的访问权限受到限制。

另一种操作系统是实时操作系统，通常用于嵌入式系统和控制、测量等应用的场合。实时的操作系统要求能够对外部的事件做出及时的反应，要求系统的响应时间短，即响应的及时性；而且要能够确保响应时间的上限，也就是响应时间的确定性。例如，实时系统中可能希望在每 50 μs 中完成一个函数的计算，这时我们要求在 50 μs 中确保这个函数的计算能够完成，而不希望这个函数有时 10 μs 就完成，而有时需要 75 μs 才完成。嵌入式的操作系统则要求系统能够在无人维护的情况下保持长期稳定的运行，要求系统的规模较小、成本较低。有些嵌入式操作系统可以根据应用的需求进行裁剪，并与应用程序一起进行编译和连接。

1.2.3. 计算机语言及其编译

计算机语言是人与计算机交流信息的表达形式。计算机的程序设计语言是编写各种计算机软件的手段，或者说是一种规范。计算机程序设计语言是用于编写软件的，它本身并不是软件。

通常可将程序设计语言分为机器语言、汇编语言和高级语言三个层次。机器语言是 一种用二进制代码表示的能够被计算机硬件直接识别和执行的语言。计算机指令的代码有一定的编码格式，有一定的构成规则。这些规则以及二进制的表达方式构成了机器语言。机器语言规定了计算机指令的编码格式，与计算机的结构有关。在不同的计算机中，机器语言一般是不同的，机器语言只是数字的集合，没有明显的特征和记忆特性，因此机器语言不便于程序员掌握和使用。

汇编语言程序采用文字符号（助记符）表示机器语言，以便于程序员记忆。汇编语言的大部分指令是与机器语言中的指令一一对应的，但不能被计算机的硬件直接识别。用户用汇编语言编写的程序，可由计算机软件将它翻译成二进制代码的机器语言，然后在计算机上运行。完成这个翻译过程的软件是汇编程序。用汇编语言或者高级语言等程序设计语言编写的程序称为源程序，存储源程序的文件称为源文件。汇编程序从源文件中读取汇编语言的程序文本，将其转换成机器语言程序后存储在一个称为目标文件的代码文件中。

计算机指令的操作码及地址码在计算机中用二进制的数据表示。这种表示方式很难被程序员阅读理解，也不便于程序员编写程序。因此通常用助记符来表示指令中的操作码和操作

数。比如用 ADD 表示加法操作，用 MOVE 表示数据移动的操作等。同样，寄存器和存储单元也可用字母以及数字构成的助记符表示。如 R_1 寄存器、R_2 寄存器、 地址为 A 的存储单元、地址为 B 的存储单元等。这样一个加法操作可以表示为

ADD A, B

计算机中所有指令的助记符的集合以及使用规则构成了汇编语言。用助记符表示的指令的使用规则称为汇编语言的词法。用汇编语言编写的程序可比较简单地转换成机器指令代码。这种转换由专门的程序完成，完成这种转换的程序称为汇编程序。汇编程序和其他程序一样，也是存储在计算机中的机器指令序列。在汇编程序运行时，它读取用户用汇编语言编写的程序，然后将这个程序转换成机器指令代码后输出。

汇编语言除了用符号来表示一些操作和存储内容外，还可以用助记符表示一个常数、一个地址码或者一个转移地址的标号等，使得程序设计方便。汇编语言程序通常还提供有关该程序装入内存中的位置的信息、表示程序段和数据段开始和结束的信息以及表示程序的开始和结束的信息等，还可以有条件汇编、文件包含、常数定义等信息。表示这些信息的指令称为伪指令（Directive），它用于向汇编程序提供必要的信息。伪指令并不转换成机器指令，而是被汇编程序过滤掉，其中的信息内容被汇编程序所采用。在汇编语言中也可以直接表示数值，数值可以是二进制、十进制或十六进制的。为了区分不同基数的数值，在汇编语言中通常用后缀字母，一般在二进制数据后加后缀 B，如 1010B；在十进制数据后可以加后缀 D，或者不加后缀；在十六进制数据后加后缀 H，如 A8H。

汇编语言中还可以有宏指令的定义。一条宏指令代表由若干条连续的指令序列。它使得汇编程序中也能够定义新的“指令”，并且使运算操作的表示更加简洁。类似于 C 语言中的宏定义，在汇编程序进行汇编的过程中要将宏指令替换为对应的指令序列，就像 C 语言中的 inline 函数一样。

与二进制的指令代码相比，汇编语言指令的可读性较好，较便于程序设计、理解和调试，但它仍然是一种面向计算机硬件的语言，程序员必须熟悉计算机硬件结构的配置、指令系统和指令编码格式。而且不同的计算机有不同的汇编语言，一台计算机上的汇编程序不能直接移植到其他计算机上运行，因此程序设计一般都采用高级语言，但高级语言编译生成的程序代码一般比用汇编程序语言设计的程序代码要长，执行的速度也慢。所以汇编语言适合于编写一些对速度和代码长度要求高的程序以及直接控制硬件的程序。

高级语言是与计算机结构无关的程序设计语言。它具有更强的表达能力，可方便地表示数据的运算和程序的控制结构，能更好地描述各种算法，而且容易学习掌握。我们通常编写计算机程序都采用高级语言，如 C 语言等。但是计算机硬件一般不能直接阅读和理解高级语言，需要由专门的软件来解决。高级语言的源程序可以通过两种方法在计算机上运行。一种是通过编译程序在运行之前将源程序转换成机器语言的程序；另一种方法是通过解释程序进行解释执行，即逐个解释并立即执行源程序的语句。编译的方法具有较高的效率，因为它只对源程序进行一次性的分析和转换，而解释执行的方法则要在每次执行时重复对源程序文本进行分析，生成可执行得代码，但有时比较困难，比如循环程序，所以编译的方法是普遍采用的方法。

过去常用的高级语言如 FORTRAN、PASCAL 和 BASIC 等。目前的高级语言正向着面向对象的方向发展，程序设计界面向可视化的方向发展，使得程序的设计效率更高、移植性更好，形成了 C++和 Java 等新型高级程序设计语言。面向对象语言使得程序员以抽象的对象进行思考，每一种对象具有一组相关的操作。新的对象类型可以在原有对象的基础上构建，

新的软件可以在已有的软件基础之上开发。面向对象高级语言的目的是使得程序员以更高的层次进行思考，并且使程序代码能够更方便地重复使用。软件代码的重复使用可大大减轻程序设计的工作量，提高软件开发效率。面向对象的语言还可以更加明确地定义软件模块之间的界面，为软件工程提供支持。Java 语言的特点是能够在各种不同的计算机上运行，并且支持面向对象的方法。Java 的编译程序将 Java 语言的源程序编译成一种称为字节代码（Bytecode）的中间代码形式，然后由计算机上的解释程序解释执行字节代码程序。

除了程序设计语言外，还有其他的计算机语言，如在各种应用程序中使用的语言。应用语言用于表示人们应用计算机完成各种任务的要求。与高级语言相比，它更加接近于人类的自然语言，更容易描述数据内容和程序内容，因而使用更加方便。常见的应用语言如数据库系统中用的查询语言（如 SQL），以及在因特网中用于描述网页的各种语言（如 XML）等。

不同层次的程序设计语言构成了计算机的不同概念模型。有了汇编语言后，汇编程序员看到的计算机就像是一个能理解汇编语言的机器，相当于在硬件的基础上建立了一个虚拟的机器。高级语言相当于在计算机上又建立了一个新的层次，高级语言的用户看到的计算机是一个可理解高级语言的机器。这个虚拟的机器与具体机器的结构无关。同样，具有 Java 解释程序的计算机构成了一个 Java 虚拟机（JVM），它能够执行 Java 的字节代码。

编译程序在对高级语言进行编译时，首先检查高级语言源程序中是否存在错误，同时对程序结构进行分析，然后将源程序转换成一种中间代码的表示形式。中间代码是一种一般化的机器码，反映出机器指令的一般特征。中间代码程序还要经过优化调整，减少不必要的指令，目的是为了改进程序运行效率、节省程序运行空间等。最后将优化后的代码转换成指定计算机的机器指令代码，这时可针对具体计算机的特征再进行代码的优化。在具有并行性开发功能的编译程序中，则还要进行程序或指令间相关性的分析，根据具体并行计算机的特征，生成高效的可并行执行的目标程序代码。

存储目标程序代码的文件称为目标文件。汇编程序能读取用汇编语言编写的源程序文件并产生一个目标文件。目标文件中包含机器指令和一些信息记录，用于将若干个目标文件组合成程序。目标文件一般是不能执行的，因为它可能引用其他程序文件中的过程以及数据。目标文件的格式与操作系统有关，通常包含一个文件头、代码段、数据段、符号表等。文件头描述文件信息，如代码段的地址、数据段的地址以及符号表地址等。符号表描述源程序中出现的各种符号（如标号）所代表的指令地址值。

图 1-4 所示是编译程序和汇编程序构成目标程序文件的例子。大多数程序由多个源文件构成，这些程序文件分别进行编写，然后进行编译或汇编。程序中还可以使用已有的实用子程序，即库程序。基本的库程序一般由程序设计语言定义，由编程或者汇编程序提供。一个程序模块中通常要引用在其他模块以及库程序中定义的子程序或数据，所以，如果一个程序模块如果引用了其他模块或库程序中的子程序或数据，那么它在汇编之后还不能执行，需要用另一个软件将相关的程序代码模块以及库程序的目标代码合成一个可执行文件，从而能够在计算机上运行。这个软件工具是连接程序（Linker），它将多个目标文件以及库文件连接成一个可执行文件，使得它们之间可以进行函数的相互调用。现在一些集成的软件开发平台包含了编译程序、汇编程序和连接程序，通常提供了源程序的编辑、编译、汇编、连接、调试等功能，能够自动地将多个源程序文件进行编译、汇编和连接操作，从而方便了程序设计。

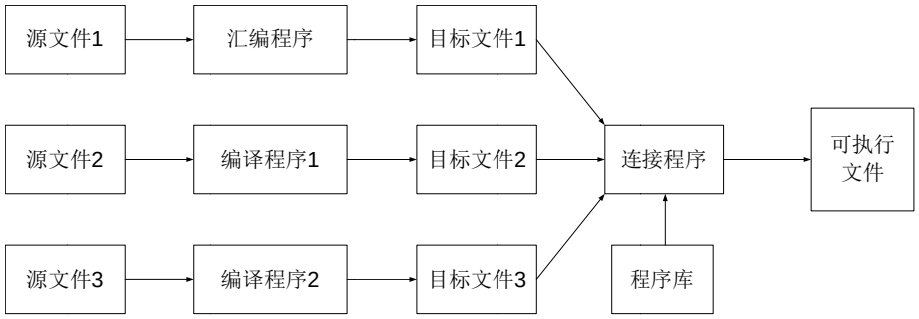


图 1-4 从多个源文件产生可执行文件的过程

将程序的源代码分成多个文件进行编译称为分离编译。分离编译是为了避免程序开发过程重复地对未修改的源代码进行编译。如果将源程序放在一个文件中，每次小小的修改就要重新编译全部源代码，包括一些从不修改的标准库。将源代码分成多个文件进行编译或汇编后，每个源文件被编译或汇编成相应的目标代码文件，需要用连接程序将多个目标代码文件连接成一个文件，即可执行文件。在分离编译构成的目标代码中，如果出现调用其他程序代码中函数的指令，或者转移到其他程序代码模块的转移指令，这些指令的目标地址是未确定的。连接程序分析各个目标代码占用的内存空间容量，计算各个函数的实际相对地址，从而确定各个函数调用的目标地址，计算各个标号的实际相对地址，从而确定各个转移指令的目标地址，将这些信息加入到各个目标代码模块中，然后将这些目标代码模块合并成一个可执行文件。可执行文件的格式一般与目标文件相同，但其中所有的目标地址都已确定下来。可执行文件在执行时需要由操作系统将其从磁盘上读入内存，然后启动它的执行。在操作系统中，这个过程是：

（1）读取可执行文件头，以确定程序代码和数据区域的大小。

（2）为代码和数据创建一个足够的地址空间。

（3）将指令和数据从可执行文件中复制到内存中。

（4）复制程序的参数（如果有的话）到主程序的堆栈中。

（5）初始化寄存器和堆栈指针。

（6）跳转到启动程序。由启动程序将参数复制到参数寄存器中，并调用程序的主函数。主函数返回时，启动程序终止该程序。

库程序中提供了一些常用的公共子程序，以方便应用程序的开发。上述程序库在连接的时候加入到可执行文件中，这种库程序称为静态连接库，因为库程序是在运行之前与应用程序连接的。静态连接库的缺点是：库函数的代码成为可执行代码的一部分，目标代码形成之后就不可能进行版本更新，而且在计算机中运行的多个应用程序中可能都包含相同的库函数代码，浪费了内存空间，为了避免这些问题，后来的操作系统中出现了动态连接库，这种库程序在运行时才按需下载并进行连接。动态连接库可以很好地支持面向对象的程序设计语言，因为面向对象的程序设计语言需要动态地下载、连接和初始化类库。

1.3. 计算机系统的历史与发展

电子数字计算机最初作为一种计算工具而问世，是人类长期努力奋斗的结果。计算的应用领域正在不断地扩展。本节简单介绍计算机的历史与发展趋势。

1.3.1. 计算机的发展历史

计算机的出现以及结构的形成是有着悠久的历史背景的。在电子计算机出现之前，人类曾经发明创造了各种各样的计算工具。早在中国的战国时期就出现筹算法。在唐朝末叶，我国人民开始使用算盘。欧洲 16 世纪出现对数计算尺。1642 年法国 Blaise Pascal 制成了世界上第一台机械计算工具，能够进行算术加减法运算。它包含表示十进制数的 6 个转盘式计数齿轮。1671 年在德国制造成了一台能够自动实现乘除法的机械计算器。后来英国人 Charles Babbage 在 1821 年和 1832 年分别制成了差分机和分析机，在分析机中采用穿孔卡片存储信息。这些计算装置的一个共同特点是在人的直接操作下进行计算。20 世纪初又出现了电动计算器。这些原始计算装置的出现为人类探索计算机原理提供了必要的基础。

电子计算机的一个重要特点是能够自动进行计算，它预先将计算的程序存储起来，并根据程序对输入的数据进行计算。电子计算机是 20 世纪出现的信息处理设备。1935 年德国的 Konrad Zuse 制造成功采用二进制算术和程序控制的机械式数字计算机 Z1，1936 年，英国数学家 Alan Turing（1912--1954）提出了一个计算机抽象模型，为计算机奠定了理论基础，这个模型称为图灵机。1938 年美国的 V.Bush 为解线性微分方程而设计的微分器是世界上第一台电子模拟计算机。这个时期，美国科学家曾研制成功机电式的计算机。到了 20 世纪 40 年代，一方面由于导弹、火箭、原子弹等现代科学技术的发展，需要解决一些极其复杂的数学问题，原有的计算工具已经不能满足要求；另一方面由于电子技术的发展，为研制电子计算机提供了物质条件。因此，电子数字计算机的出现已成为历史的必然。

1943 年开始研制的 ENIAC（Electronic Numerical Integrator And Calculator）是美国第一台由程序控制的电子数字计算机，由宾夕法尼亚大学的 J.W.Mauchley 和 J.P.Ecker 和莫尔小组（Moore School）为进行新武器的弹道计算而制造。该计算机曾在第二次世界大战中曾投入使用，到 1946 年正式公布。它包含 18800 个真空电子管和 1500 多个继电器，每秒可进行 5000 次加法。存储容量为 20 个 10 位的累加器，用线路连接方式建立数据通路和编排程序。机器占地 170m^2，长 30m，重约 30 吨，耗电 150kW。与此同时，英国研制的电子计算机在 1943 年投入运行，用于破译德军的电报密码。同年，美国的哈佛大学与公司合作生产了采用电磁继电器的计算机。

ENIAC 是美国的第一台全电子的计算机，但 ENIAC 还不是存储程序的计算机。也就是说，计算程序不是存储在存储器中的，程序的改变靠人工设置开关和插拔电线，用改变接线的方法进行。ENIAC 并没有采用二进制的计算方式，而是采用十进制的计算方式。它采用十条信号线来表示 0--9 之间的一个数据。

1950 年第一台存储程序计算机 EDVAC（Electronic Discrete Variable Automatic Computer）诞生，由匈牙利数学家冯.诺依曼与莫尔学院合作研制。该计算机有一个大得多的存储器，它由容量为 1014 字的水银延迟线主存储器和容量为 20KB 的磁线辅助存储器组成。EDVAC 计算机还采用二进制的数据表示方法，并设置了能对二进制数进行运算的 ALU。该计算机采用存储程序工作方式，运算程序存储在内存中，可以随时修改，该计算机的结构采用五大模块的结构。存储程序的计算机把指令和数据放在同一个存储器系统中，必要时可以将指令作为数据处理。存储程序的工作方式使得计算机能够完成各种所需的功能，只要这种功能能够用算法描述，从而成为一台“万能的”数字计算设备。存储程序的计算机工作方式源于 1940 年英国普林斯顿的一家研究所， 1948 年在计算机中首次得到实现。存储程序使得计算机的程序可以灵活改变，从而产生了软件的概念。

此后，IBM 公司也生产了一系列的大型计算机系统，用于在计算中心供大量用户使用，包括早期的电子管计算机和 360 系列集成电路计算机。20 世纪 50 年代末美国的 DEC 公司开始生产小型计算机产品，用于办公环境和工业生产过程控制。1970 年代末 则出现了微型计算机，形成桌上型计算机。1976 年 Cray 公司研制成功 Cray-1 超级计算机，用于解决复杂的计算问题。20 世纪 80 年代还出现了具有较强图形功能的工作站型计算机和用于在网络环境中提供网络通信和磁盘存储等功能的服务器型计算机系统。IBM 公司的第一台个人计算机（PC 机）采用 16 位的微处理器和 64KB 存储器以及一个低密度的软盘驱动器，没有图形显示功能。

从 1977 年以来，一枚集成电路芯片中所能集成的晶体管数量迅速增长，从几百个晶体管到一亿个晶体管以上，基本上每三年翻两番。近几十年来国外电子计算机发展的重大事件如表 1-1 所示。

表 1-1　国外电子计算机发展大事记

年份	大事
1938	Konrad Zuse 建成第一台二进制的机电式通用计算机 Z-1
1943	Alan Turing 等建成了一台正空管计算机
1945	J.W.Mauchley 教授等研制成 ENIAC
1947	由于 IBM 公司和哈佛大学共同研制成自动机电式哈佛 Mark I 计算机
1948	曼彻斯特 Mark I 成为第一台存储程序的数字计算机
1952	EVDAC 研制成功
1952	IBM 制成第一台军用的存储程序电子计算机 IBM 701
1954	Univac1103A 成为第一台商业计算机，采用磁芯存储器
1956	采用晶体管的 Univac 商用计算机开发成功
1960	DEC 公司 11 月研制成 PDP-1，第一台具有显示器和键盘的商用计算机
1961	IBM 公司研制成 7030，号称超级计算机
1962	英国研制成 Atlas 计算机，首次采用虚拟存储器和流水操作
1964	IBM 宣布 System/360
1964	CDC6600 研制成功，第一台商用超级计算机
1965	DEC 推出 PDP-8，采用晶体管线路
1968	Seymour Cray 设计成功 CDC7600 超级计算机，40MFLOPS
1971	Intel 推出第一个微处理器芯片 4004
1972	DEC 推出 PDP-11
1975	第一台微型机 Alter 8800 研制成功
1976	Cray-1 研制成功，第一台向量结构超级计算机
1977	Tony 和 Commodore 推出商品微型机
1980	Apollo 公司研制成第一台工程工作站
1981	IBM 推出 PC 机
1982	Cray X-MP 推出将两台 Cray-1 连接在一起
1982	日本启动“第五代”计算机项目
1985	Cray-2 和 Connection Machine 研制成功，性能均达每秒十亿次运算
1989	Cray-3 研制成功，采用砷化镓芯片
1991	Cray Y-MP C90 研制成功，采用 16 个处理机

在几十年的发展过程中，计算机经历了电子管、晶体管、集成电路、大规模集成电路和

超大规模集成电路等发展阶段，计算机系统的性能迅速提高，价格迅速下降，体积不断缩小，耗电不断下降，可靠性不断提高。计算机的应用领域不断扩展，从最初的科学计算发展到数据处理、实时控制、辅助设计、通信、教学以及人们的日常生活等领域。以后还出现了嵌入式计算机，将计算机系统集成到各种机器、仪表、家用电器和移动设备中。计算机的发展在很大程度上受到电子技术发展的推动，而计算机的发展也促进了电子工业特别是微电子工业的发展。

将计算机分成五个模块的实现方式最早是由冯.诺依曼提出的，称为冯.诺依曼结构。冯.诺依曼结构还确定了计算机二进制数据的表示方式，以及存储程序的工作方式。在计算机完成指定功能之前，先将实现该功能的程序（即软件）装入内存，然后根据程序的规定一步一步地完成操作。计算机中存储的程序和数据都是二进制形式的代码。

早期的计算机没有复杂的操作系统，当时计算机一次只能运行一个程序，在这样的系统里，计算机操作者必须进行大量复杂的操作，计算机的运行速度经常受到操作员工作速度的限制而处于等待状态。在后来的系统中，人们开发出一些控制程序，操作者可以用一组操作命令对系统进行控制，如程序文件的分配、磁带系统的装卸、程序运行的起始与结束。操作系统作为用户与硬件的接口，还简化了文件的创建和执行。

操作系统曾经经历了批处理系统、多道程序系统和实时系统等发展阶段。早期的计算机系统与用户是隔离的，程序和信息以穿孔卡片和纸带的形式送入机房，由计算机操作员将计算任务成批地输入计算机处理，并将输出结果以打印的形式分发给用户。这种做法称为“批处理”。由于输入输出速度较低，为了提高主机的工作效率，采用了多套输入输出设备同时操作的系统，主机能够同时处理多个用户提交的程序。这样构成了多道程序系统。实时操作系统是具有及时响应能力的操作系统，早期曾用于飞机订票系统和工业控制系统等。

计算机高级语言是用于进行程序设计的计算机语言，所以也是一种程序设计语言。计算机高级语言有许多种，但得到广泛使用的只有十几种，如 FORTRAN、COBOL、BASIC、PASCAL、C 等。FORTRAN 是 1956 年出现的，由 IBM 公司研制而成，是一种公式翻译语言，有丰富的标准库，适合于科学计算。Algol 是 1960 年出现的，是一种适合于描述数值计算过程的语言。COBOL 出现于 1959 年，是面向商业的通用语言。C 语言具有一些较低层的编程功能，适合于系统程序的设计。BASIC 语言原先是一种小型通用语言，适合于初学者学习，现在也已发展成一个较复杂的程序设计语言。

1.3.2. 计算机的分类

计算机按照其软硬件系统开放程度可分为通用计算机和专用计算机。目前的通用计算机系统一般都是采用超大规模的通用 CPU 芯片构成的，提供标准的总线插槽方便内存和外设的扩展；软件系统通常基于开放的操作系统平台，如 DOS、Windows 和 Linux，用户可根据需要安装各类应用程序。通用计算机系统开放性好，操作图形界面友好，但是可能对于某一特定的任务，它的工作效率不是最佳的。专用计算机则是针对某一特定领域设计的系统，针对特定的计算和控制任务进行了优化，如针对数字信号处理应用的数字信号处理器（DSP，Digital Signal Processor）的和针对网络处理应用的网络处理器（NP，Network Processor）等。目前在工业控制、仪器仪表、家用消费电子产品中广泛应用的嵌入式计算机就是属于专用计算机。

根据计算机的性能和应用特征，也可将计算机分为桌上型计算机、服务器型计算机和嵌入式计算机。桌上型计算机包括 PC 机、工作站和笔记本型计算机，为一个用户提供良好的

计算性能和较低的成本。桌上型计算机是成本低、应用广的计算机类型。过去，PC 机和工作站的主要区别主要在于系统的配置和运行的操作系统，工作站的配置比 PC 机高。但是现在随着机的大量应用，促进了它的发展速度，现在 PC 机的配置逐渐与工作站接近，两者的性能差别逐渐消失。笔记本电脑（Laptop Microcomputer）又称为膝上型计算机，它的功能基本上与桌上微型计算机（DeskTop Microcomputer）相同，Laptop 在体积、重量、低功耗等方面要求适应便携和电池供电的需要。

桌上型计算机系统中除了 CPU 和存储器芯片外，还有大规模集成的接口芯片，用于连接 CPU、存储器、图形接口、磁盘接口，以及各种串行口、并行口、输入输出总线等。桌上型计算机系统的通常要求具有合理的成本和较高的性能，如 Intel 公司的奔腾 D 系列 CPU 就是为桌上型计算机设计的 CPU 产品。一般台式计算机可以允许较高的耗电功率和风扇方式的散热，而笔记本计算机则因为采用电池供电和体积较小，要求耗电省而且发热量较小。

图 1-5 是采用奔腾 D 的桌上型 PC 机的结构例子。其中 82955 是支持奔腾 D 至尊版处理器与图形控制器和主存之间的接口芯片，它支持 8GB/s 的图形接口 PCI-Ex16，支持两个 DDR2 SDRAM 接口；82801GR 是奔腾 D 处理器与外围设备的接口，它支持 PCI-E 总线、PCI 总线、USB 串行口、串行 ATA（SATA）的 IDE 磁盘接口。

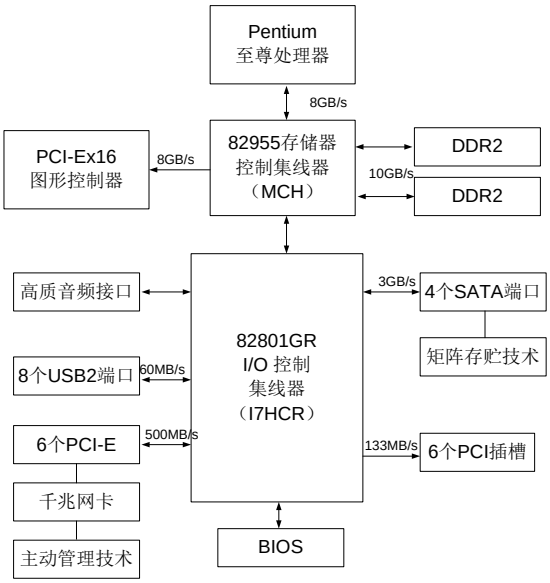


图 2-5 PC 机构成

服务器型计算机是用于在计算机网络上提供文件存储和共享、打印、通信等服务的计算机设备，它不仅需要具备较高的运算性能，还要求具备很大的存储容量、很大的输入输出带宽，要求具有很好的可靠性和可用性，能够不间断地提供服务。为了实现高可靠性和高可用性，要求网络服务器具有子系统管理的功能，能够进行故障记录和故障报告，以便于系统管理员了解系统的可靠性情况，及时更换可能出现问题的部件。服务器系统一般应具有一定的冗余，以保证系统出现故障时的正常运行。此外，网络服务器还要求具有性能的可升级性，能够通过增加或者更新部分部件提高系统的性能。一般的台式计算机难以达到上述要求。在服务器中采用的 CPU 一般为 64 位高档处理器。在 64 位 CPU 的系统结构中，定义了 64 位的数据格式和指令，具有 64 位虚拟存储器地址空间。

服务器的类型又可以分为超级服务器、大型服务器和小型服务器等类型，分别相当于过去概念中的超级计算机、大型计算机、小型计算机。超级计算机（Supercomputer）主要用

于科学计算领域，它是专用于解决科学计算中特殊问题的机型，又称巨型机。超级计算机中都采用最新的技术，以提高运算速度和数据存储能力，但对输入输出能力的要求相对较低。

超级计算机一般是专供研究人员使用的，而不是被许多用户共同使用的。从运算性能上讲，超级计算机一般比同时期的其他类型的计算机要高。近年出现的超级计算机通常采用并行处理技术，将多台计算机组织成一个功能强大的系统。大型计算机（Mainframe）适合于作为企业级的计算应用，一般具有大容量存储设备和网络连接能力。小型计算机（Minicomputer）的特点是体积小、成本低，可以为少量人员提供计算和存储服务，适合于小型企业和工作组等小型应用场合。现在的各种服务器在网络提供相同的功能，它们可以采用与微型计算机相同的技术实现，但是具有更大的可扩展性，包括计算能力、存储能力和通信能力等方面。

除了采用容错技术外，服务器系统还可采用集群（Cluster）技术提高性能和可靠性。集群技术是通过网络将计算机连接成一个并行系统的技术。服务器采用集群技术后形成物理上连接并紧密集成的两台或多台服务器，形成一个统一的整体，进行系统级的容错。集群中的各个处理机可以不完全相同，每台处理机都能独立完成特定的任务。构成一个集群服务器系统需要特殊的软件、硬件支持。集群配置可以保证当某台处理机或应用程序发生意外故障时，集群中的另外一台处理机可以在继续自己份内工作的同时，接过发生故障服务器上的任务。集群技术还可以实现系统的可伸缩性：即系统中可以增加处理机数量，并且随着处理机数量的增加，系统的性能也相应提高的特性。在集群系统中，如果系统负载超出了系统的承受能力，可向现有系统内增加结点（处理机），系统中多台服务器可以执行同样的应用和数据库操作，从而提高整体性能。

在网络服务器中为了提高磁盘存储器的容量及其可靠性，通常还采用磁盘阵列，或者称为独立冗余磁盘阵列（RAID）。它采用多个低成本的硬盘，构成一个磁盘阵列， 数据展开存储在多台磁盘上进行并行的读写操作，这样提高了数据传输的带宽和磁盘操作的处理能力。磁盘阵列具有容量大、数据传输速率高、可靠性高等优点，可应用于服务器等需要大容量存储的计算机系统中。磁盘阵列的数据并行存储操作对 CPU 是透明的，多个磁盘驱动器在逻辑上构成一个磁盘，系统的数据分布在各磁盘上的操作由磁盘阵列控制器完成。

1.4. 嵌入式计算机

在更多的应用中，计算机作为一个部件成为其他设备的一部分，如作为机器人的大脑或者作为家用电器的控制部件。这种计算机称为嵌入式的计算机，它的成本更低，用途更广。嵌入式计算机的结构通常是面向特定应用的，为特定应用专门开发设计的。不同的嵌入式应用有不同的要求，差异较大，需要根据不同的应用进行专门的开发设计。一般的嵌入式计算机的硬件包括微处理器、存储器及 I/O 接口、图形控制器等，它以通用微处理器作为内核，集成其他接口电路和存储器接口。如果在嵌入式微处理器中集成了存储器电路，则构成了单片机。

嵌入式微处理器一般对实时多任务有很强的支持能力，能完成多任务并且有较快的响应时间，要求应用程序代码和实时内核的执行时间减少到最低限度。

嵌入式微处理器一般具有可扩展的处理器结构，从而能迅速地开发出满足应用的嵌入式系统。此外，嵌入式微处理器的功耗很低，尤其是用于便携式的无线及移动的计算和通信设备中靠电池供电的嵌入式系统更是如此。嵌人式系统设计通常采用片上系统（System on Chip，SoC）的设计方式。系统芯片有各种满足片上功能的嵌入式 IP 核（Intellectual Property core）

组合而成。这些 IP 核包括微处理器、存储器、音频和视频控制器、调制解调器、USB 功能模块、DSP 模块等。嵌入式计算机的特点是：软硬件系统经过裁剪，专为某个特定的应用而设计，CPU 只运行一个应用程序；采用短小轻型的操作系统，可减少对存储器容量的需求，并且具有更佳的实时性能；某些嵌入式微处理器内嵌数字信号处理部件。

嵌入式计算机的软件与桌上型计算机不同，嵌入式计算机的软件不是存储在磁盘文件中，而是固化在 ROM 中。ROM 包含了嵌入式计算机的操作系统软件和应用软件，这种程序代码称为映像（Image）。现在，许多嵌入式系统都采用闪存（Flash）来代替传统的 ROM 存储器或者磁盘，用于存储最终开发形成的程序代码映像和运行中构成的数据。嵌入式系统中的实时操作系统现在也普遍采用多任务的操作系统，能够提供任务调度、任务间的通信和同步机制。多任务的环境使得系统能够实现多种功能，任务间的通信和同步机制用于实现任务间的数据交换的运行协调关系。这里的任务可以是一个进程，也可以是一个线程。作为实时操作系统，还需要有一个定时器，以保证系统在一定的延迟时间内对外部事件做出反应。嵌入式实时操作系统软件同样也分为内核和外围两个层次，内核提供任务间的同步、调度和定时机制等，外围模块提供一些与外围硬件有关的功能，包括存储器管理、各种输入/输出接口和设备的管理等。

嵌入式计算机在应用数量上远远超过了各种通用计算机，一台通用计算机巧外部设备中就包含了多个嵌入式微处理器，键盘、鼠标、软驱、硬盘、显示卡、显示器、网卡、调制解调器（Modem）、声卡、小型计算机系统接口（Small Computer System Interface，SCSI）、打印机、扫描仪、USB 集线器等均是由嵌入式处理器控制的。

图 1-6 所示的是一个嵌入式处理器应用的例子。它是一个掌上型计算机。该计算机采用的 CPU 是 StorngARM SA1100 处理器，主频为 50--206MHz，具有耗电省的特点。其中，坞站是用于与 PC 机连接的底座，IrDA （Infrared Data Association）是连接红外设备的数据接口标准。子板接口提供各种扩展功能。可连接的子板包括存储器板、与 PC 机连接的 PCMCIA（Personal Computer Memory Card International Association）接口、USB 串行口、UART（Universal Asynchronous Receiver/Transmitter）串符口、同步数据链路口 SDLC（Synchronous Data Link Control）、同步串行口 SSP（Synchronous Serial Port）、远程 通信的编码/解码器（Codec，或 Coder/Decoder）等。

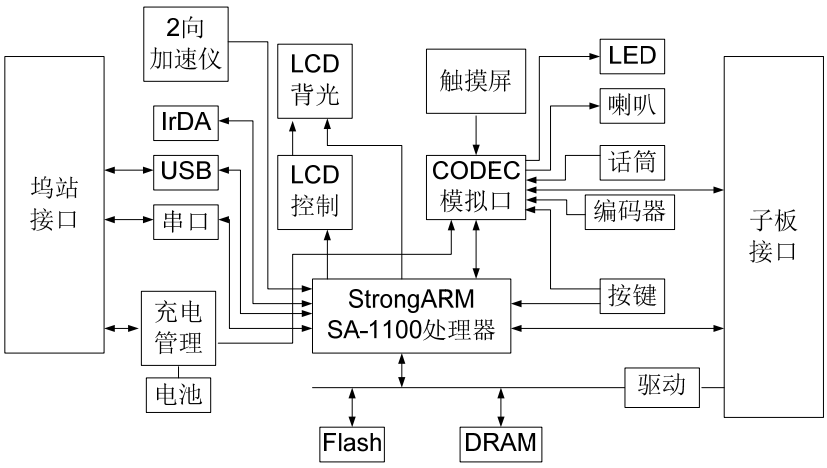


图 1-6 嵌入式系统的例子

嵌入式计算机系统现已在工业测控、消费电子产品、家用电器、仪器仪表以及军事等广

泛领域得到广泛应用，提高了人们的生活质量和工作效率。许多嵌入式应用要求采用 SoC 的技术，因为只有当各种功集成在一块芯片上时，才能达到应用的低成本、低功耗和体积小的要求。嵌入式计算机的应用领域包括以下方面：

信息电器（Information Appliance）。信息电器是一种嵌入式的计算装置。具有移动性和简单易用的用户界面，具有多种应用功能，包括 Web 浏览、电子邮件、可视电话、电子书籍、网络游戏、个人数字助理、家用电器控制等功能，构成未来的移动计算系统。

移动设备，如个人数字助理（PDA，Personal Digital Assistant）、智能手机、数字相机、商务通、条码扫描器等。这种嵌入式处理器通常要求具有功耗低的特点。嵌入式微处理器最大并且增长最快的市场是手持设备、电子记事本、手机等消费类电子产品。移动计算对数字系统提出了功能多、体积小、耗电省、价格低、重量轻等要求。

交互式数字媒体，如数字机顶盒、交互式电视机、视频游戏机等。

嵌入式控制设备。如网络设备、办公自动化设备、通信设备、存储设备（如 RAID，Redundant Array of Independent Disk，控制器）等。网络设备包括路由器、网桥、交换机、网卡。办公自动化设备包括打印机、扫描仪、复印机等。通信设备包括公共小交换机 PBX（Private Branch Exchange）、蜂窝机站等。

嵌入式计算机系统的另一个主要应用是控制应用和测试应用，实现工业设备以及各种家用电器的工作控制，包括电视机、洗衣机、电冰箱等。这种应用场合要求具有实时性。在制造工业、过程控制、通讯、仪器、仪表、汽车、船舶、航空、航天、军事装备、消费类产品等方面均是嵌入式计算机的控制应用领域。采用嵌人式计算机的自动测试设备可对测试数据进行分析和转换，形成智能化的测试仪器。

桌上型和嵌入式计算机的一个重要应用功能是数字信号处理（DSP，Digital Signal Processing），而其核心就是数字信号处理器（DSP，Digital Signal Processor）。数字信号处理是利用计算机或专用处理设备，以数字形式对信号进行采集、变换、滤波、估值、增强、压缩、识别等处理，以得到符合人们需要的信号形式。数字信号处理可用于音频数据处理中的音效处理功能，可用于对外围设备的状态和控制信号的处理等。在 DSP 系统中，输入信号是代表各种物理量的模拟量电子信号。输入信号首先进行带限滤波和采样，然后进行 A/D（Analog to Digital）变换，将信号变换成数字数据流。DSP 处理器对输入的数字信号进行处理。经过处理后的数字量值再经 D/A（Digital to Analog）变换，转换成模拟量，之后再进行内插和平滑滤波，就可得到连续的模拟量的波形。DSP 处理的数据是对表示成数字量序列的信号进行信号滤波、时域频域转换等处理。由于 DSP 处理器可以对数据进行任意的计算处理，可以实现各种要求的信号变换，所以能实规模拟量信号的电路中无法实现的处理方式。DSP 计算的任务主要是进行语音和音频压缩、滤波、频谱分析、调制/解调、纠错码的编码和解码、伺服控制、音频处理、语音识别、信号合成等。其中 DSP 的音频处理能够实现音频的频谱改善、变速播放、混音处理、环绕声处理和谐音处理等。目前 DSP 广泛应用于蜂窝电话机、硬盘驱动器、调制解调器、图像压缩器（如数字相机和数字电视机中）、语音编码器、音频处理器等，此外还在个人通信设备、电机控制、 定位系统、视频会议、安全通信等产品中得到应用。

数字信号处理与一般数据的运算处理尽管存在许多相似的地方，但也还是存在许多差别。DSP 要求完成信号的实时处理，运算速度是首要的指标。为了提高速度，通常将指令存储器和数据存储器分开，构成哈佛结构，以加快访存速度。DSP 的特点是重复地进行相同的数字计算，处理的程序相对较简单，主要是乘法和累加运算；DSP 计算过程中需要进行频

繁的数据访问，访问的数据是不断到达的信号序列（信息流）；数据的处理要求在一定的时间范围内完成，对数据进行实时的处理。随着 DSP 应用的不断普及，一些通用计算机中也增加了支持 DSP 的功能，以加快 DSP 的速度。

1.5. 数据编码

数据在计算机中需要进行各种方式的编码，以便于数据的存储和处理，不同类型的数据在计算机中需要采用不同的编码表示，计算机的运算过程就是对用二进制编码的数据进行处理存储的过程，是计算机的基本功能。

在计算机中，数据的表示有多种形式，如将数据分为定点数据、浮点数据、文字数据、音频数据、视频和图形数据等，这些数据的编码构成计算机的基本数据编码。为了保证计算机数据在运算和传输过程中的可靠性，计算机中还广泛采用检错码和纠错码。在选择计算机的数据编码方法时，要考虑的是数据的类型、数值的范围、数值的精确度、数据存储和处理所需要的硬件代价等。在处理数据编码时，目标之一是用尽可能少的代码来表示各种信息，以减少不必要的代码存储和计算开销。数据编码还要求代码的格式具有规整性，以方便代码的运算处理以及今后发展、升级。本章介绍计算机中常见的数据编码方法。

1.5.1. 数制的转换

我们日常生活中都习惯采用十进制数进行计数和计算。十进制数的表示有以下的规则：即在表示任何数值时，除了表示正负的符号外，只用 0—9 这 10 个数字符号，表示大于 9 的数时用多个数字符号的排列表示，处于不同位置上的数字字符表示不同的数值，这个数值是数字符号乘以某个权数的结果，该权数在十进制数中等于 10 的指数。数字符号的位置称为位序号 n，个位的位序号为 0，十位数的位序号为 1，以此类推，权数的值等于 10 的 n 次幂。多位数据中每一位记数符号代表的数值是记数符号乘以 10 的指数的结果。指数的值等于位序号。一个数字的实际值等于各位上的实际值的总和。例如，对于十进制数 123，三个数据位分别代表 100、20 和 3。其数值可表示为

$$1\times10^{2}+2\times10^{1}+3\times10^{0}$$

表达式中取 10 的幂次是因为它代表十进制数，指数的值取决于数据位序号，即个位取 0，十位取 1，百位取 2，以此类推。一般地，对于一个 n 位的十进制数 $x_0x_1...x_{n-2}x_{n-1}$，它代表的数值可表示为

$$x_0 10^{n-1}+x_1 10^{n-2}+\cdots+x_{n-2}10^{1}+x_{n-1}10^{0}$$

其中，对于各位数字 x_i（$0\le i\le n-1$），有 $0\le x_i\le 9$。

十进制数的这种计数方法可以推广到其他的计数制，基于同样的原理，我们可以定义其他的进位制。一般地，在 r 进制下，数 $x_0x_1...x_{n-2}x_{n-1}$ 所代表的值可表示为

$$x_0 r^{n-1} + x_1 r^{n-2} + \cdots + x_{n-2} r^1 + x_{n-1} r^0$$

其中，r 称为基数，各位数字 x_i（$0 \le i \le n-1$）取值范围在 0 到 r-1 之间。

计算机中采用的二进制数采用 0 和 1 两个符号表示数值。表示大于 2 的数时用多个数符号的排列表示，每个记数符号也称为一个位。多位数据中每一位记数符号代表的值是记数符号乘以 2 的指数的结果。指数的值也等于位序号。一个多位数据的值同样于各位上的实际值的总和。例如，二进制数 1101_2 (下标 2 表示二进制）所代表的数用十进制数表示可写成

$$1\times 2^3 + 1\times 2^2 + 1 = 8 + 4 + 1$$

一般地，对于二进制的定点整数 $x_0 x_1 \ldots x_{n-2} x_{n-1}$，它代表的数值为

$$x_0 2^{n-1} + x_1 2^{n-2} + \cdots + x_{n-2} 2^1 + x_{n-1} 2^0$$

为了区别二进制数与十进制数，通常在二进制数中用下标 2 表示。十进制数是常用表示方式，下标可以省略。二进制数可以方便地在计算机中进行存储和计算，但它表示数据需要用较多的位数，不便于人们书写和记忆。为此，在计算机中还常常把若干个二进制数位组合起来，构成八进制数或十六进制数等。因为 3 个二进制位正好构成一个八进制 4 个二进制位正好构成一个十六进制位。八进制数采用 0—7 这 8 个数字符号。多位数据中每一位记数符号代表的数值是记数符号乘以 8 的指数的结果。在十六进制数 除了采用 0—9 这 10 个符号外，还采用字母 A—F 这六个符号表示一位中大于 9 的数。即 A 表示十进制的 10，即 10_{10}，B 表示 11_{10}，……，F 表示 15_{10}。

八进制数和十六进制数与二进制数的转换十分方便，可以采用位段转换的方法。在将二进制数转换成八进制数时，因为 3 位二进制数正好对应 1 位八进制数，所以可以从小数点的位置开始，将其三位一组进行转换。如果要将二进制数转换成十六进制数时，则四位一组进行转换。二进制数与十六进制数的转换关系如下所示。

$$\begin{array}{llll} 0000_2 = 0_{16} & 0001_2 = 1_{16} & 0010_2 = 2_{16} & 0011_2 = 3_{16} \\ 0100_2 = 4_{16} & 0101_2 = 5_{16} & 0110_2 = 6_{16} & 0111_2 = 7_{16} \\ 1000_2 = 8_{16} & 1001_2 = 9_{16} & 1010_2 = \mathrm{A}_{16} & 1011_2 = \mathrm{B}_{16} \\ 1100_2 = \mathrm{C}_{16} & 1101_2 = \mathrm{D}_{16} & 1110_2 = \mathrm{E}_{16} & 1111_2 = \mathrm{F}_{16} \end{array}$$

【例 1-1】将 1110100102 转换成八进制数和十六进制数。

解：将数据转换成八进制数时，从低位到高位将二进制数分为 111，010，010 三组,其中 1112 转换成 78，0102 转换成 28，将结果连起来得到 7228，即 1110100102=7228。

在将二进制数据转换成十六进制数时，同样从低位到高位将二进制数分组，剩下一组不足 4 位则补 0 计算，得 00012，11012，00102，最低四位 00102 转换 216，次低四位 11012 转换成 D16，高四位 00012 则转换为 116，最后得到的结果为 1D216，即 1110100102=1D216。

小数部分的八进制数和十六进制数与二进制数的转换与此类似。要注意的是在分组时要从小数点位置开始进行分组，也就是要从小数点开始从高位到低位进行分组，最后剩余位数不够一组的位数时补 0。

十进制数到二进制数的转换可分为整数和小数两种情况。整数的十进制数到二进制数的转换可以采用除 2 取余的方法。即把十进制数除以 2，所得余数作为二进制数的最低位，再除以 2，所得余数作为次低位，如此重复，直到商数为零为止。然后将余数连起来形成数据的二进制表示。因为当我们把一个数值看成是二进制数时，它的表达式为上述多项式，数制的转换也就变成了求这个多项式中各个系数的值。为了求得这个表达式的系数，只要将这个表达式除以 2 并取余数即可。第一次除以 2 得到的余数是 x_{n-1}，第二次除以 2 得到的余数是 x_{n-2}，以此类推。

以上主要讲了整数的数制及其转换，对于带小数的数据，我们可以把它分成整数部分和小数部分，然后分别进行转换。任意进制的小数同样可以表示为基数的幂次多项式。对于一个 n 位的十进制纯小数 $x_0.x_1\cdots x_{n-2}x_{n-1}$，它代表的数值可表示为

$$x_0 10^0 + x_1 10^{-1} + \cdots + x_{n-2} 10^{-(n-2)} + x_{n-1} 10^{-(n-1)}$$

其中，对于各位数字 x_i（$0 \le i \le n-1$），有 $0 \le x_i \le 9$。

同样，我们可以定义其他进位制小数的权值表示。在 r 进制下，数 $x_0.x_1\cdots x_{n-2}x_{n-1}$ 所代表的值可表示为

$$x_0 r^0 + x_1 r^{-1} + \cdots + x_{n-2} r^{-(n-2)} + x_{n-1} r^{-(n-1)}$$

其中，r 称为基数，各位数字 x_i（$0 \le i \le n-1$）的取值范围在 0 到 r-1 之间。

对于带有小数的十进制数，在转换成二进制数时，则必须对小数部分采用按乘 2 取整数的方法。其规则是：用 2 乘以待转换的十进制数的小数部分，取乘积的整数部分作为转换后的二进制数中小数部分的最高位数字；再用 2 乘以上一步乘积的小数部分，再取新乘积的整数部分作为转换后二进制小数低一位数字；不断重复直至乘积部分为 0 或得到的小数的位数满足要求时结束。然后将整数连起来形成数据的二进制表示。因为纯小数的二进制表达式为

$$x_0 2^0 + x_1 2^{-1} + \cdots + x_{n-2} 2^{-(n-2)} + x_{n-1} 2^{-(n-1)}$$

其中 $x_0 = 0$，为了求得 $x_1, \cdots, x_{n-2}, x_{n-1}$ 的值，我们可以将这个数不断乘以 2 并取其整数部分。

对于既有整数部分又有小数部分的十进制数，可以先转换其整数部分，再转换其小数部分。将得到的两部分合并起来就得到了转换后的带小数的二进制数。

将十进制数转换成八进制数或十六进制数同样可以采用除八取余或除十六取余的方法，但这两种除法不便于计算。我们可以先将十进制数转换成二进制数，再将二进制数转换成八进制数或者十六进制数。将十进制小数转换成八进制小数或十六进制小数同样可先将十进制小数转换成二进制小数，然后转换成八进制小数或十六进制小数。

八进制数或十六进制数转换成十进制数则可以采用上述按权相加的方法。

【例 1-2】将 167_8 和 $1C4_{16}$ 转换成十进制表示。

解：将数据的各位按权相加

$$167_8 = 1\times8^2 + 6\times8 + 7 = 64 + 48 + 7 = 119_{10}$$
$$1C4_{16} = 1\times16^2 + 12\times16 + 4 = 256 + 192 + 4 = 452_{10}$$

1.5.2. 定点数的编码

在计算机中用编码表示的二进制数据称为机器数。它是一个位数固定而且数值量有限的二进制代码，因此表示的数值范围也是有限的。计算机中表示的机器数是一系列离散的数据。一个机器数所代表的实际数值称为真值。机器数有带符号数和无符号数两种。带符号数中一般用最高位表示数据的正负符号。无符号数的每一位都代表数值，只能表示正数和零，通常直接用它的二进制数作为机器数。

1.5.2.1. 无符号数的编码

定点数是指小数点位置固定不变的数据。在计算机中，小数点的位置是约定的，并不需要用代码表示。通常采用两种类型的定点数表示。一种定点数把小数点定在最低位数的右面，这种定点数称为定点整数，因为它实际上没有小数点。另一种定点数把小数点固定在最高位数的后面，即纯小数表示，称为定点小数，它只保留小数点后面的数据位，小数点前面的一位数据固定为零。二进制定点数可以直接在计算机中表示，计算机中表示无符号数就直接用这种二进制的表示作为数据的编码（机器数)。

对于一个 n+1 位二进制的定点整数 $x = x_0x_1x_2\cdots x_n$。其中 $x_i = 0$或1，$0 \le i \le n$， 这个数代表的数值为

$$x_0 2^n + x_1 2^{n-1} + \cdots + x_{n-1} 2^1 + x_n$$

它可表示的数值范围是

$$0 \le x \le 2^{n+1} - 1$$

定点小数的数据只表示数据的小数部分，把小数点固定在最高位的后面，最高位为 0，即纯小数表示的数据。对于一个 n+1 位的定点小数 $x = x_0.x_1x_2\cdots x_n$，其中 $x_i = 0$或1，$0 < i \le n$，$x_0 = 0$，这个数代表的数值为

$$x_1 2^{-1} + \cdots + x_{n-1} 2^{-n+1} + x_n 2^{-n}$$

它可表示的数值的范围为 0--1-2^{-n}，即

$$0 \le x \le 1 - 2^{-n}$$

在采用这种数据表示方法的计算机系统中，大于 1 的数据必须先通过合适的比例因子转换成小于 1 的数，并保证运算中的中间结果数据也都小于 1，而在输出结果时再将数据按比例放大。这种表示方法主要用在早期的计算机中，它最节省硬件。现在的计算机中一般都可表示多种数据类型，包括定点数和浮点数。在定点数据编码中表示带有小数的数据时，并不需要用一个代码来表示小数点的位置，而只需约定小数点的位置，因为这个小数点的位置在运算过程中是不变的。

计算机中的数据编码都是有一定的表示范围的、离散的，而不像数学中的数，可以是任意大的、连续的。在数据编码中，如果数据值超过了编码所能表示的数值范围，则称为数值溢出。在定点整数中，数据编码的位数越多，则数据表示的个数越多，数据表示的范围越大，为了避免定点整数的溢出，可以增加数据编码的位数，从而扩大数据表示的范围，在定点小数中，数据编码的位数越多，则数据表示的个数越多，精度也越高。

1.5.2.2. 有符号数的编码

在计算机中，为了表示负数，需要在数据表示中增加表示正负符号的信息。为此可以增加一个符号位，但是仅仅增加符号位还不够，还要考虑负数中其余位的编码方法和运算方法，以便于数据的计算。计算机中表示一个带符号数的方法有原码表示法、反码表示法、补码表示法和移码表示法四种，它们构成四种定点数的编码方法。

1. 原码

原码表示法中用一个符号位表示数据的正负，用 0 代表正号，1 代表负号，其余的代码表示数据的绝对值，称为数值位。我们用 x 表示数据的真值，用$[x]_{原}$表示数据在计算机中的原码表示法的编码。对于一个 n+1 位的二进制定点整数$x = x_0.x_1x_2 \cdots x_n$，原码表示的定义是

$$[x]_{原} = \begin{cases} x, & 0 \le x < 2^n \\ 2^n - x = 2^n + |x|, & -2^n < x \le 0 \end{cases}$$

即当$x \ge 0$时，$[x]_{原}$与 x 的二进制表示形式上一样，为$0x_1x_2 \cdots x_n$（因为$x \le 2^n$，所以$x_0 = 0$）；当$x < 0$时，$[x]_{原}$的代码与二进制数$2^n - x$的表示形式相同，是$1x_1x_2 \cdots x_n$（因为$2^n - x > 2^n$，所以$x_0 = 1$。这里的最高位x_0就是符号位。一个正整数的原码需要在它的二进制表示中加上一个符号位 0，一个负整数的原码需要在它的二进制表示中加上一个符号

位 1。例如，当年 n=4 时，设 x=1010，则$[x]_{原}$=01010；如果 x=-1010， 则$[x]_{原}$=11010。

对于给定的定点整数原码$[x]_{原}$，它的数值 x 可根据以下公式求得

$$x=(-1)^{x_0}(x_1 2^{n-1}+\cdots+x_{n-1}2+x_n)$$

一个 n+1 位定点整数原码的数值范围是

$$-2^n+1\le x\le 2^n-1$$

对于定点小数，若数据的二进制表示为$x=x_0.x_1x_2\cdots x_n$，则原码表示的定义是

$$[x]_{原}=\begin{cases}x, & 0\le x<1\\ 1-x=1+|x|, & -1<x\le 0\end{cases}$$

对于给定的定点小数原码$[x]_{原}$的代码，它的数值为

$$x=(-1)^{x_0}(x_1 2^{-1}+\cdots+x_{n-1}2^{-n+1}+x_n 2^{-n})$$

一个 n+1 位定点小数原码的数值范围为

$$-1+2^{-n}\le x\le 1-2^{-n}$$

在原码表示中，零有两种表示方式。即 000…0 和 100…0。原码表示的优点是简单易懂，乘除法运算的规则比较简单，但它的缺点是加减运算的实现比较复杂。在两数相加时，需要对符号位进行判断，如果是同号则进行加法运算；如果是异号则要进行减法运算。而在进行减法操作时，还要比较绝对值的大小，然后用绝对值大的数减去绝对值小的数，再确定结果的符号位。

2. 补码

补码表示法也是用最高一位代表符号，其余各位代码表示数值。用表示数据在计算机中的补码表示法的编码。对于一个 n+1 位的二进制定点整数$x=x_0x_1x_2\cdots x_n$， 补码的表示方式定义为

$$[x]_{补}=\begin{cases}x, & 0\le x<2^n\\ 2^{n+1}+x, & -2^n\le x<0\end{cases}$$

正数的补码与原码相同，负数的补码是等于 x 加上2^{n+1}的二进制编码表示（因为

$-2^n \le x \le 0$时，$2^n \le x + 2^{n+1} < 2^{n+1}$，所以$x_0 = 1$。对负数的补码编码相当于将 x 对$2^{n+1}$求模所得的二进制编码表示。例如，当 n=4 时，设 x=1010，则$[x]_{补}$=01010； 如果 y=-1010，则$[y]_{补}$=100000-1010=10110。在 8 位的计算机中，数据都用 8 位的编码表示，这时数据位有 7 位，即 n=7， $[x]_{补}$=00001010， $[y]_{补}$==100000000-00001010=11110110。

对于给定的定点整数补码$[x]_{补}$的代码，其数值为

$$x = -x_0 2^n + x_1 2^{n-1} + \cdots + x_{n-1} 2 + x_n$$

一个 n+1 位定点整数补码的数值范围是

$$-2^n \le x \le 2^n - 1$$

求一个数的补码在计算机中通常采用简便的方法。对于正数，直接取其二进制数的表示构成补码。对于负数的补码可有两种方法，一种方法是在对其按位取反之后再在最低位加 1；另一种方法是从最低位开始，对遇到的 0 和第一个 1 取其原来的代码，从第一个 1 以后开始直到最高位，均按位取其相反的代码（即求非）。后一种方法的电路比较简单。

对于定点小数$x = x_0.x_1x_2\cdots x_n$，其补码的定义为

$$[x]_{补} = \begin{cases} x, & 0 \le x < 1 \\ 2 + x, & -1 \le x < 0 \end{cases}$$

这里，负数的补码是将 x 对 2 求模所得的二进制编码表示，所以称为模 2 补码。对于给定的定点小数补码$[x]_{补}$，它的数值为

$$x = -x_0 2^0 + x_1 2^{-1} + \cdots + x_{n-1} 2^{-(n-1)} + x_n 2^{-n}$$

一个 n+1 位定点小数补码的数值范围为

$$-1 \le x \le 1 - 2^{-n}$$

在补码表示中，负数比正数多一个，零有唯一的编码，即 000…0。

将一个数的补码按位取反，在最低位上加 1 后就得到它的相反数的补码。我们经常用这种方法来求得一个负数补码的值。我们把对数据的编码按位取反，在最低位上加 1 的操作称为求补操作。这样，根据$[x]_{补}$求$[-x]_{补}$的方法可表示为是对$[x]_{补}$进行一次求补的操作。

对一个数的补码进行求补操作后，得到这个数的相反数的补码。也就是说，已知$[x]_{补}$，求$[-x]_{补}$的方法是对$[x]_{补}$进行一次求补操作，而不管 x 是正数还是负数。

补码的特点是加减法运算简单，不论是正数还是负数，运算方法都是相同的。在确定数据的编码表示时，要考虑的一个重要问题是数据运算的实现，因此在计算机中通常采用补码来表示数据。

为了便于判别运算是否溢出，某些计算机中还采用一种双符号位的补码表示方式，称为模4补码，因为它相当于数据对4取模的结果。对于一个n+1位的定点整型数$x = x_0x_1x_2\cdots x_n$，模 4 补码的表示方式定义为 n+2 位：

$$[x]_{补} = \begin{cases} x, & 0 \le x < 2^n \\ 2^{n+2} + x, & -2^n \le x < 0 \end{cases}$$

对于正数，其模 4 补码的两个符号位为 00；对于负数，其模 4 补码的两个符号位为 11。n+2 位模 4 补码的数值范围和 n+1 位模 2 补码的相同，即

$$-2^n \le x \le 2^n - 1$$

对于定点小数$x = x_0.x_1x_2\cdots x_n$，其模 4 补码的定义为

$$[x]_{补} = \begin{cases} x, & 0 \le x < 1 \\ 4 + x, & -1 \le x < 0 \end{cases}$$

3. 反码

反码的最高位也是符号位，0 表示正数，1 表示负数。对于 n+1 位的二进制定点整数$x = x_0x_1x_2\cdots x_n$，反码表示的定义是

$$[x]_{反} = \begin{cases} x, & 0 \le x < 2^n \\ 2^{n+1} - 1 + x, & -2^n < x \le 0 \end{cases}$$

在二进制数中，$2^{n+1}-1$表示为全 1，所以反码也称为 1 的补码。例如，当 n=4 时，设 x=1010，则$[x]_{反}$=01010；如果 x=-1010，则$[x]_{反}$= 10101。对于给定的定点整数反码$[x]_{反}$的代码，其数值为

$$x = -x_0(2^n - 1) + x_1 2^{n-1} + \cdots + x_{n-1}2 + x_n$$

对于正数，反码表示与原码和补码相同，直接在二进制数前加上符号位 0 构成。负数的反码是在对数据的二进制表示按位取反后得到。一个 n 十 1 位定点整数反码的数值范围是

$$-2^n+1 \le x \le 2^n-1$$

对于定点小数 $x = x_0x_1x_2\cdots x_n$，反码的定义是

$$x_n[x]_{反} = \begin{cases} x, & 0 \le x < 1 \\ 2-2^{-n}+x, & -1 < x \le 0 \end{cases}$$

其中，$2-2^{-n} = 1.11111\cdots1$，即全 1，加上负值的 x 后得到的二进制位与 x 的每一位都相反。对于给定的定点小数反码的$[x]_{反}$代码，它的数值为

$$x = -x_0(1-2^{-n}) + x_1 2^{-1} + \cdots + x_n 2^{-n}$$

其数值范围为

$$-1+2^{-n} \le x \le 1-2^{-n}$$

在反码表示中，零有两个编码，即 000…0 和 111…1。由于反码的运算不方便，所以在计算机的数值计算中，很少得到实际应用。但是我们可以利用反码的计算方法求数据的补码。因为我们比较反码与补码的公式，可见当 x<0 时，

$$[x]_{反} = 2-2^{-n}+x$$

$$[x]_{补} = 2+x$$

可得到

$$[x]_{补} = [x]_{反} + 2^{-n}$$

这就是通过反码求补码的公式。即求一个负数的杯码的方法是先求其反码，然后在最低位上加 1。

4.移码

原码、反码和补码编码方法都有一个缺点，就是不便于比较数据的大小。例如在补码中，两个正数的比较是容易进行的，我们可以得出 001000〈001001，前者的值是 8，后者的值是 9。但是补码不便于比较正数与负数之间的大小，例如 111111<000001，前者的值是-1，后者的值是 1。为了解决这个问题，引入了移码的表示法。移码编码的方法是给数据加上一个常数，使得负数变成正数，取其二进制表示构成数据的编码。这个常数通常是 2 的幂次。

对于 n+1 位二进制整数 $x = x_0 x_1 x_2 \cdots x_n$，移码的定义为

$$[x]_{移} = 2^n + x, -2^n \le x \le 2^n$$

其数值范围为

$$-2^n \le x \le 2^n - 1$$

例如，当 n=4 时，设 x=1010，则 $[x]_{移} = 11010$；如果 x=-1010，则 $[x]_{移} = 00110$。在上述移码的定义中，最高位仍可看做是符号位，但 1 表示正号，0 代表负号。 将移码编码与补码编码进行比较，我们可以发现它们的符号位相反，数值位相同。对移码表示的数据可以进行加减运算。但对移码数据进行加减运算时需要对结果进行修正， 修正量为 2^n，也就是将符号位取反。移码表示中，0 有唯一的编码，即 100…0。

移码的另一个特点是：移码的编码保持了数据原有的大小顺序，这样便于进行比较操作。因此它被用于表示浮点数的小数点位置，因为在浮点数的加减运算中，首先需要对两个数据的小数点位置进行比较。由于移码在编码中只是加了一个常数，所以它仍然保持了数据之间的大小关系不变。这个特征使得移码便于进行数据比较操作。

定点小数没有移码定义。表 1-2 中列出了-8--7 之间的数在计算机中的 4 种不同编码。从表中可见，原码、反码和补码这三种数据表示方法的共同点是：编码的最左位为 1 时表示一个负数，为 0 时表示一个正数。而移码则与此相反，1 表示正数，0 表示负数。表中还可看出移码保持数值的大小顺序不变。

数据真值的二进制表示与机器数很容易混淆，它们的一个区别是，数据的真值中位数是不限的，不存在溢出问题，可以把最高位的 0 省略，如二进制数 001011 可以写成 1011。原码、补码等都是机器数。机器数的位数是有规定的，不能忽略任何位置上的 0 或 1，如果把 001011 写成 1011，就不再满足位数要求，也改变了符号位。数据编码有一定的表示范围和精度，编码的位数不同，它代表的数据范围或精度也不同。

表 1-2

数值 ：	原码	反码	补码	移码
-8	—		1000	0000
-7	1111	1000	1001	0001
−6	1110	1001	1010	0010
-5	1101	1010	1011	0011
-4	1100	1011	1100	0100
-3	1011	1100	1101	0101
-2	1010	1101	1110	0110
-1	1001	1110	1111	0111
-0	0000	1111	0000	1000
0	1000	0000		
1	0001	0001	0001	1001
2	0010	0010	0010	1010
3	0011	0011	0011	1011
4	0100	0100	0100	1100
5	0101	0101	0101	1101
6	0110	0110	0110	1110
7	0111	0111	0111	1111

数据真值与机器数的另一个区别是，带符号的机器数的最高位代表数据的符号，数据真值的最高位则不代表符号。在表示数据真值时可以在它的前面加上正负符号，如+123、-32等；但是在机器数编码之前不能加这种符号，因为它的最高位已经表示了数据的正负符号。

在计算机中定点数据的编码一般是由软件完成并存入存储器中的，硬件通常不需要提供编码方面的支持，只需要提供运算方面的支持。

1.5.3. 浮点数的编码

1.5.3.1. 浮点数的规格化表示方法

在定点数据编码中存在的一个问题是难以表示数值很大的数据和数值很小的数据。这是因为小数点只能固定在某一个位置上。这就限制了数据表示的范围，为了表示更大范围的数据，数学上通常采用科学记数法，把数据表示成一个小数乘以一个以 10 为底的指数。在计算机的数据编码中可以把表示这种数据的代码分成两段，一段表示数据的有效值部分,另一

段表示指数部分，也就是表示小数点的位置在这种表示方法中，小数点的位置是可以浮动的，因此称为浮点数。

在浮点数的编码中，数据代码分为符号位 S(Sign)、尾数 M（Mantissa）和阶码 E(Exponent)三部分。尾数以定点小数编码的形式表示数据值；阶码则是指数部分的编码，代表小数点的位置。符号位表示整个数据的正负。这样一个浮点数编码所代表的数值为

$$N = (-1)^s \times M \times R^E$$

其中，R（radix）是基数。基数及一般为 2、8 或者 16，基数在系统中是固定的，数据编码中不需要用代码表示。

在浮点数编码中，符号位为 0 表示正数，为 1 表示负数。尾数似采用定点小数的编码形式表示，可以用原码或补码编码方式，它的位数决定了浮点数的表示精度。阶码是一个整数的编码，它的位数决定了数据的表示的范围。为了便于运算过程中比较数值的大小，阶码部分一般采用移码表示。在浮点数据的表示方法中，符号位为 0 表示正数，为 1 表示负数。浮点数的编码格式如图 1-7 所示，阶码在符号位之间。例如，当阶码采用 3 位移码，尾数采用 4 位编码时，$\left[0.1010\times 2^2\right]_{浮} = 01101010$。

符号	阶码	尾数

图 1-7 浮点数的编码格式

在这种浮点数表示方法中，阶码加 1 或者减 1 使得数据值增加到原来的只倍或者 1/R 倍，相当于把尾数的小数点右移或者左移一位，所以同一个数据可以有多种编码表示。为了使浮点数编码具有唯一性，以及在尾数中表示最多的有效数据位，浮点数的编码都采用规格化的表示方法。规格化的浮点数编码规定尾数部分用纯小数形式给出，而且尾数的绝对值应大于或等于 1/R。对于二进制尾数的情况，规格化要求尾数的绝对值大于或等于 1/2，并且小于或者等于 1。这样，当尾数与符号位采用原码编码时，尾数的最高位应当是 1。当尾数与符号位采用补码编码时，为了便于判别，规定尾数的最高位与符号位相反。这样，对于用补码表示的尾数，其规格化数值范围为$1/2 \le M < 1$及$-1 \le M < -1/2$。不符合这种规定的数据可通过修改阶码并同时移动尾数的办法使其满足这种格式。

在浮点数中，基数影响到数据的精度和范围，由于尾数部分在机器中最终仍然表示为二进制代码，一位基数为 R 的尾数表示为若干位二进制数的一组代码。这一组代码作为一个整体进行操作。这样，基数影响到尾数部分小数点移动的单位,也影响到规格化的定义。如果基数为 2，那么当阶码加 1 或减 1 时，尾数的小数点应右移或左移一位 以保持浮点数的数值不变；如果基数为 16，那么当阶码加 1 或减 1 时，尾数的小数点就应移动 4 位（16 进制的 1 位），以保持数值不变。当尾数采用原码表示时，如果基数为 2，那么 0.1000 是规格化的尾数，0.0001 则不是规格化的尾数；如果基数为 16，那。么 0.0001 是规格化的尾数，因为它大于或等于 1/16。对于基数为 16 的浮点数，规格化时要求尾数小数点后第一个 16 进制位数据不为零，这样尾数小数点后第一个 16 进制位就可以取 1--F 之间的数，也就是二进制 0001～1111 之间的代码。

当浮点数的尾数部分的数值为 0 时，不论其阶码为何值，都是零值。为了使得零的表示的唯一性，规定阶码也必须为零，称为机器零。机器零是一个特殊的合法浮点数编码，尽管

不符合浮点数规格化表示的要求。

浮点数编码方法表示的数据是离散的数值，而不是连续的实数。它扩大了数值表示的范围，但并未增加数值表示的个数。例如，对于以 R 为基数，有 1 位符号位、p 位阶码和 m 位二进制尾数代码的浮点数，如果阶码采用移码表示，那么它可表示的最小的规格化尾数值为 1/R，最大的尾数值为 1-2^{-m}，最大阶码值为$2^{p-1}-1$，最小阶码值为-2^{p-1}。可表示的最小正值为$1/R\times R^{-2^{p-1}}=R^{-2^{p-1}-1}$、可表示的最大值为$R^{2^{p-1}-1}(1-2^{-m})$，可表示的规格化尾数个数为$2^m\times(R-1)/R$。加上不同阶码的表示和零的表示， 浮点数可表示的数据个数为$2^{p+m+1}(R-1)/R+1$。一般来说，定点数编码方式的运算比较容易实现，但它表示的数值范围有限。而浮点数表示的数值范围很大，但运算电路的实现十分复杂。

在浮点数编码中，当数据的绝对值太大，以至于大于阶码所能表示的数据时，称为浮点数的上溢（Overflow)。而当数据的绝对值太小，以至于小于阶码所能表示的数据时，则称为浮点数的下溢（Underflow)。浮点数的溢出主要是因为阶码出现了溢出。在数轴上，浮点数的数值表示范围以及溢出数值范围如图 1-8 所示。浮点数编码所表示的数据在数轴上的分布不是等距离的。越靠近原点，数据分布越密集；越远离原点，分布越稀疏。在浮点数据格式的设计中，有一个在数据表示范围和数据表示精度之间进行权衡的问题。对于一定编码长度的浮点数据，阶码位数越多，数据表示的范围就越大，但是减少了尾数的位数就使得数据表示的精度下降。增大基数可以扩大数据表示的范围，但是因为所表示的数据总数是有限的，所以可表示数据的间隔也会增大，这就意味着数据表示精度的下降。

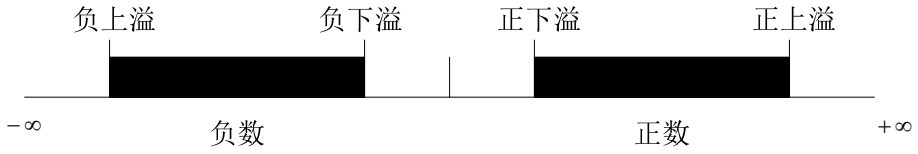


图 1-8 浮点数的表示范围

【例 1-3】浮点数的编码。对数据 123 作规格化浮点数的编码，假定 1 位符号位， 基数为 2，阶码 5 位，采用移码，尾数 10 位，采用补码。

解：

(1)先将数据表示成$M\times R^E$的形式

$$123_{10}=1111011=0.1111011000\times 2^7$$

分别对上述结果按照题目要求进行编码，阶码采用 5 位移码，尾数采用 10 位补码。

$$[7]_{移}=10000+00111=10111$$
$$[0.1111011000]_{补}=0.1111011000$$

将编码的结果按浮点数格式表示出来。因为符号位为 0，所以浮点格式为

0 10111 1111011000

【例 1-4】设浮点数的格式中有 1 位符号位，6 位阶码，采用移码表示，9 位尾数，与符号位一起采用规格化的原码表示，基数为 2。问：

(1)它能表示的数值范围是什么？

(2)它能表示的最接近于 0 的正数和负数分别是什么？

(3)它共能表示多少个数值？

结果用十进制数 2 的幂次表示。

解:

9 位尾数可表示的最小正尾数为2^{-1}，最大正尾数为$1-2^{-9}$。

9 位尾数可表示的最小负尾数为$-1+2^{-9}$，最大负尾数为-2^{-1}。

6 位阶码采用移码编码后可表示的最小阶码为$-2^5=-32$，最大阶码为$2^5-1=31$，

因此：

(1)数值范围为$-2^{31}(1-2^{-9})$到$2^{31}(1-2^{-9})$。

(2)最接近于 0 的负数为:$-2^{-1}\times 2^{-32}=-2^{-33}$；

最接近于 0 的正数为：2^{-33}。

(3)符号位有 2 种取值，6 位阶码有2^6种取值。9 位尾数在规格化要求下，最高为 1，所以有$1/2\times 2^9$种取值，加上机器零，共能表示$2\times 2^6\times 1/2\times 2^9+1=2^7\times 2^8=2^{15}+1$个数值。

1.5.3.2. IEEE754

浮点数的格式已经标准化。目前，广泛采用的浮点数据编码的标准是所谓 IEEE754 标准。在这个标准中，利用基数为 2 的规格化原码编码的尾数中的第一位为 1 的特点，在尾数中设置一个缺省的 1。在 IEEE754 标准中还对浮点运算中的各种可能出现的情况作了完整的定义，定义了一些特殊的数据格式以处理上溢、下溢等异常情况。例如在上溢结果的操作中可以将结果设置为表示正无穷大或者负无穷大的代码。阶码中保留了一个编码值用于表示这种特殊的数据。标准中还定义了表示无定义数据的代码（NAN，Not A Number），如发生在零除以零、对负数求平方根或者无穷大除以无穷大时生成的结果。此外，标准中还允许有非格式化的数据。在数据代码中阶码和尾数各占用的位数有多种不同的分割方法，在 IEEE754 浮点数标准中，定义的浮点数的格式如表 1-3 所示。其中浮点数有 32 位、64 位和 80 位三

种格式，分别称为短实数、长实数和临时实数，短实数和长实数又分别称为单精度数和双精度数。临时实数用于运算中的中间结果的表示。

表 1-3 IEEE754 浮点数格式

浮点数	符号位	阶码	尾数	总位数
短实数	1	8	23	32
长实数	1	11	52	64
临时实数	1	15	64	80

IEEE754 浮点数编码格式定义如表 1-4 和表 1-5 所示。其中 S 表示发信号的无定义。

表 1-4 IEEE 单精度浮点数编码格式

符号位	阶码	尾数	表示
0/1	255	非 0	S
0/1	255	非 0	Q
0	255	0	+INF
1	255	0	-INF
0/1	1-254	f	$(-1)^s \times 1.f \times 2^{e-127}$
0/1	0	f(非 0)	$(-1)^s \times 0.f \times 2^{e-127}$
0/1	0	0	+0/-0

数据（signaling NAN)，表示需要进行处理的运算异常,Q 表示不发信号的无定义数据(quiet NAN)，表示可不进行处理的异常。表中 s 是符号位，f 为尾数，e 为阶码，INF 为无穷大。标准格式化浮点数的数值范围如表 1-5 所示。其中有些编码的规定如下：

表 1-5 IEEE 双精度浮点数编码格式

符号位	阶码	尾数	表示
0/1	2047	非 0	s
0/1	2047	非 0	Q
0	2047	0	+INF
1	2047	0	-INF
0/1	1-2046	f	$(-1)^s \times 1.f \times 2^{e-1023}$
0/1	0	f(非 0)	$(-1)^s \times 0.f \times 2^{e-1023}$
0/1	0	0	+0/-0

表 1-6 IEEE754 浮点数的数值范围

格式	最小值	最大值
单精度	2^{-126}	$2^{-127} \times (2-2^{-23})$
双精度	2^{-1022}	$2^{1023} \times (2-2^{-52})$

（1）无定义数 NAN，NAN 表示一个非数字值。发信号的无定义数 S 的尾数最高有效位为 0；不发信号的无定义数 Q 的尾数最高有效位为 1。S 作为算术操作的输入时， 结果是 Q，而 Q 作为输入时结果总是 0。引进无定义数据表示的目的是检测非初始化值的使用。程序员或编译程序可以用无定义数据表示每个变量的非初始化值。

浮点操作正常数据输入时在一定情况下也会产生无定义数据。定义 NAN 的代码可以使计算机在出现异常时能够继续进行下去。由于 NAN 的尾数段是非定义的，这就使程序可以给不同变量赋予统一的 NAN 编码。而且 NAN 输人的尾数段在 NAN 输出时是复制的，于是 就可以通过查看算法的最后 NAN 结果，以确定结果的正确性和问题的来源。没有数学解释的操作（例如 0/0）将产生一个非数字值 NAN。

（2）无穷大数 INF。引入无穷大数的目的与引入 NAN 值的目的是一样的，使计算能在异常情况下继续进行，同时为程序提供了检测错误的能力。正无穷大是大于每个有限数的，负无穷大是小于每个有限数的。NAN 与无穷大都是操作的有效输入值，也都是操作的可能输出结果。

（3）零。零有两种编码表示:+0 与－0。差不多在所有情况下，它们是等效的。然而，舍入方式可以影响到一个零的结果是正或是负，而且有些操作受其来源值的符号影响，这时 +0 和-0 之间就有微小的差别。

（4）规格化数。对于任意的规格化数采用阶码与尾数来编码。在实数与长实数格式中作为其尾数的最高有效部分假定有一个隐含的位，这就给出了一个额外的有效位。

（5）非规格化数（Denormal），或者叫做子规格化数（Subnormal）。这些数据代码的阶码部分为零，即表示最大的负数数值，而尾数的高位部分有一个或多个连续的 0。非规格化数用于表示某些下溢数据，它的大小在零与最小有限数之间。标准的一个重要特点就是利用这种编码来实现逐级下溢概念，即当结果比可表示的规格化数还小时，程序可继续运行下去。例如当一个系统的最小规格化数为0.1×10^{-99}时，可以用以下对十进制数据运算结果的处理表示这种逐级下溢的方法（用箭头表示舍入）：

$$①2.50000\times10^{-60}\times3.50000\times10^{-43}=8.75000\times10^{-103}\rightarrow0.00088\times10^{-99}$$
$$②2.50000\times10^{-60}\times3.50000\times10^{-60}=8.75000\times10^{-120}\rightarrow0.0$$
$$③5.67834\times10^{-97}-5.67812\times10^{-97}=2.20000\times10^{-101}\rightarrow0.02200\times10^{-99}$$

即当计算结果的阶码太小而无法用规格化数表达时，则对这个结果进行调整，直到阶码成为最小规格化数的指数时为止；然后对该数进行舍入并存储。而当结果太小以至于非规格化数也无法表示时，则将其规格化为零。

IEEE754 标准还规定了临时实数的操作，使其数值范围与精度大于基本格式要求， 但不是采用双倍位数表示中间结果，而是借助减少舍入误差来提高最后结果的精度，提供扩展精度的临时实数是为了获得更高精度的结果，而又不会增加通常与更高精度有关的运算延迟。

【例 1-7】将十进制数-0.75 表示成单精度的 IEEE754 标准代码。

解：-0.75 可表示成-3/4。即二进制的-0. 11。在浮点表示法中为

$$-0.11\times2^{0}$$

在 IEEE754 规格化的表示法为

$$-1.1\times2^{-1}$$

根据 IEEE754 的单精度表示公式

$$(-1)^{s}\times1.f\times2^{e-127}$$

这个数可表示为

$$(-1)^{1}\times(1+0.1000\ 0000\ 0000\ 0000\ 0000\ 0000)\times2^{-1}$$
$$=(-1)^{1}\times(1+0.1000\ 0000\ 0000\ 0000\ 0000\ 0000)\times2^{126-127}$$

即

1 01111110 10000000000000000000000

【例 1-8】将如下 IEEE754 单精度浮点数用十进制数表示：

1 10000001 01000000000000000000000

解：符号位 S=1，阶码部分值 e=129，尾数部分义二 1/4 = 0.25，根据 IEEE754 表示公式

$$
\begin{aligned}
&(-1)^1 \times (1+0.25) \times 2^{129-127} \\
&= -1 \times 1.25 \times 2^2 \\
&= -1.25 \times 4 \\
&= -5.0
\end{aligned}
$$

1.5.4. 文字的编码

计算机通常需要对文字信息进行处理，如人的姓名、地名和各种文档等。文字信息包括各种语言的文字，这里只介绍英文和中文信息的编码。

1.5.4.1. 英文字符的编码

对英语文字的编码通常用 7 位或者 8 位二进制的数据表示一些字母、数字符号、标点符号和一些控制符号等，通常采用一个字节表示一个字符信息。英文字符的编码方案有多种，目前国际上普遍采用的一种字符编码系统是 ASCII 码（American Standard Code for Information Interchange)。在这种编码标准中规定 8 个二进制位的最高一位为 0，余下的 7 位可以给出 128 个编码，表示 128 个不同的字符，编码范围是 00_{16} -- $7F_{16}$ 。其中的 95 个编码对应着英文字母、数字等可显示和可打印的字符。另外的 33 个字符的编码值为 0--31 和 127，表示一些不可显示的控制字符。代码定义如表 3-6 所示，其中 b_0 ～ b_6 、分别表示代码的第 0 位到第 6 位，第 7 位代码 b_7 恒为 0。其中，各控制字符代表的意义如表 3-7 所示。

ASCII 码是 7 位的编码，但由于字节是计算机中存储信息的基本单位，因此一般仍以一个字节来存放一个 ASCII 字符。ASCII 编码已被国际标准化组织 ISO 和国际电报电话咨询委员会 CCITT(现改为国际电信联合会 ITU）采纳而成为一种国际通用的信息交换用标准代码。

表 1-8 ASCII 码表

$b_6b_5b_4$ / $b_3b_2b_1b_0$	000	001	010	011	100	101	110	111
0000	NUL	DLE	SP	0	@	P		p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	“	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	•	7	G	W	g	w
1000	BS	CAN	（	8	H	X	h	x
1001	HT	EM	）	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	；	K	[	k	{
1100	FF	FS	，	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	-	o	DEL

计算机的键盘能够将用户的击键用 ASCII 编码的形式传送给主机。计算机运算器中对于 ASCII 编码的文字信息的处理主要是字符串的比较、插入、删除等，从而支持对文字信息的更复杂的处理。一些计算机中提供了直接对 ASCII 字符的操作指令，能够对字符串进行比较、插入、删除等操作。

表 1-9 ASCII 控制字符表

控制字符	说明	控制字符	说明
NUL	空	SOH	标题开始
STX	文本开始	ETX	文本结束
EOT	发送结束	ENQ	询问
ACK	应答	BEL	响铃
BS	退格（backspace）	HT	横向制表
LF	换行(line feed)	VT	纵向制表
FF	格式走纸	CR	回车(carriage return）
SO	移出(shift out)	SI	移入(shift in）
DLE	数据连接交换	DC1-DC4	设备控制
NAK	否定回答	SYNC	同步(synchronous)
ETB	信息发送组结束	CAN	删除
EM	媒体结束	SUB	代替
ESC	换码（伪鄉已）	FS	文件分隔
GS	组分隔	RS	单位分隔
SP	空格（space）	DEL	删除（delete）

1.5.4.2. 汉字的编码

中文字符的存储和输入要比英文字符复杂得多。因为汉字的数量很多，在进行编码时我们既要考虑编码的紧凑性以减少存储量，又要考虑输入的方便性。中文的信息中通常还包含着英文的字符，所以汉字的编码也应包含英文字符的编码。在汉字中将输入用的编码和计算机内存储的编码分开定义。目前，计算机中的汉字编码都是软件定义和处理的。硬件不直接提供对汉字处理的支持。

计算机的键盘是为西文输入设计的，为了利用西文键盘输入汉字，需要建立汉字与键盘按键的对应规则，将每个汉字用二组键盘按键表示。这样形成的汉字编码称为汉字的输入码。汉字输入码应当规则简单、容易记忆，同时为了提高输入速度，输入码的编码应尽可能短。常见的汉字输入码有数字编码、拼音码和字形码等。数字码的如国标区位码，它的特点是无重码，每个编码对应唯一个汉字。拼音码根据汉字的拼音规则进行编码，具有简单易记的优点，缺点是重码多，因为汉字中有许多同音字。字形码的典型例子是五笔字型编码，它根据汉字的笔画规则进行编码。计算机的汉字输入软件将键盘的输入码转换成汉字的机内码，并存储在存储器中。

机内码是汉字在计算机内部进行存储和处理时采用的表示形式，它同样是一种二进制代

码。汉字机内码是用于汉字信息存储、交换、检索等操作的内部代码。1980 年公布的国家标准 2312 规定了 3755 个最常用汉字和 3008 个较常用汉字的编码。根据这个标准，把这 6763 个汉字分成若干个区，每区包含 94 个汉字，采用两个字节表示一个汉字。每个汉字的编码由两部分组成，第一部分（首字节）指明该汉字所在的区，第二部分（尾字节）指明它在区中的位置。其中，首字节编码为 81_{16}—FE_{16}，以便与 ASCII 码区别。

目前最新的汉字编码是 2000 年公布的国家标准 GB18030，该标准收录了 27484 个汉字，总编码空间超过 150 万个码位，为解决人名、地名用字问题提供了方案，为汉字研究、古籍整理等领域提供了统一的信息平台基础。新的汉字编码标准采用单字节、双字节和四字节三种方式对字符编码，全面兼容 GB 2312。单字节部分使用 00_{16}—$7F_{16}$ 码字（对应于 ASCII 码的相应编码）。双字节部分，首字节编码从 81_{16}—FE_{16}，尾字节编码分别是 40_{16}--72_{16} 和 80_{16}--FE_{16}。四字节编码在第二个字节部分采用 30_{16}--39_{16} 之间的编码作为对双字节编码扩充的后缀，这样扩充的四字节编码的范围为 81308130_{16}--$FE39FE39_{16}$。

国际上广泛采用的另一种汉字编码是 Unicode。它是一种 16 位的文字编码方案，与 ASCII 编码兼容。它能够包含世界各个国家文字中主要字符的编码，同时还定义了扩展编码的方案。在 Unicode 中，规定从 0000_{16}—$1FFF_{16}$ 间的 8192 个编码用于拉丁文、 阿拉伯文、希伯来文、古埃及文、古斯拉夫文、希腊文、非洲文、东南亚等地区文字的编码，从 2000_{16}—$2FFFF_{16}$ 之间的 4096 个编码用于货币符号和各种数学符号的编码，从 3000_{16}--$3FFF_{16}$ 之间的 4096 个编码用于汉文、日文、韩文拼音字母和标点符号的编码， 从 4000_{16}—$4FFF_{16}$ 的 4096 个编码用于统一的汉文、日文、韩文、印度文字符的编码，从 $E000_{16}$—$EFFF_{16}$ 之间的 4096 个编码用于扩展编码，从 $F000_{16}$—$FFFF_{16}$ 之间的 4096 个编码为保留编码。Unicode 已成为 Java 语言的默认字符集，其具体编码方案可以从它的网站（http://www.unicode.org/）得到。

1.5.4.3. 十进制数的编码

在计算机中，一般是把十进制数据转换成二进制数进行处理的。在计算机中用硬件直接处理十进制数的原因是：在一些商业计算等计算机应用场合中，直接处理十进制形式的数据在某些方面有其优点，可以避免将十进制数转换成二进制数的操作，所以有些计算机中提供了十进制数的直接编码和处理方式。十进制数据在计算机中主要有两种表示形式：

(1）字符串形式，即用一个字节的编码表示一个十进制的数位或符号。用连续的多个字节表示一个完整的十进制数据。

（2）压缩的十进制数串形式，即用一个字节的编码表示两个十进制的数位。它比较节省存储空间，又便于进行算术运算。

字符串形式表示数据的常见方法是用 ASCII 代码表示数据。这种用字符构成的数字字符串在计算机的某些应用领域中使用得十分频繁，如在金融、商业应用和统计工作中。因此有些计算机可以直接处理数字字符串，也就是在计算机中直接表示和处理十进制的数据，这样可提高数据的表示范围和运算精度，可方便地控制运算的精度。

在 ASCII 代码中可以表示数字字符串，0--9 的数字在 ASCII 字符码中的代码分别 0110000--0111001，其低四位等于数字的原码。这样用一个字节表示一个十进制的数字，一个十进制的数据用若干字节的一个字符串表示。为了表示数据的符号以及对数据字符串进行分隔，可以采用前分隔字符串和后嵌入字符串两种方式。

前分隔字符串是让符号位占用单独的一个字节，并把符号位放在数字位之前，用字符“+”

表示正号，用字符“-”表示负号。后嵌入数字串则不为符号位单独分配一个字节，而是把它嵌入到最低位数字中。其规则是，把负号变成十六进制的 40，并将其与最低数字位的值相加。这种表示比前分隔字符串少用了一个字节，但使得最低位数字位的编码与其他数字位的编码方法不一致。

上述字符串形式的数字表示方法中的冗余较大，在表示十进制数的一个字节 4 位的数据在运算时不具有数字的意义；因此这种表示方式主要用于非数值运算的场合。在进行数值运算中还可采用压缩的十进制表示方式，即在一个字节中存放两个十进制的数字。这种方式称为二一十进制表示方式，或二一十进制码(BCD 码)。它可节省存储空间，又便于直接处理十进制的数据，是广泛采用的一种数据表示方法。但是这种表示仍然有一定的代码冗余，与二进制编码的运算部件相比，二一十进制的运算器电路仍然比较复杂。

为了表示带符号数，在二一十进制数的字符串中可以用一个表示符号的代码，它也是一个 4 位的代码，放在最低数据位之后。符号代码的值可以采用 4 位编码中的剩余 6 个代码之一。如用 1100_2 表示正号，用 1101_2 表示负号。在这种表示方式中，数字位数加符号位必须为偶数，对于奇数位的情况应在最高位之前补一个数字 0。

十进制数的这种编码也由软件实现，不需要硬件提供编码方面的支持。由于这种编码与 ASCII 编码十分相似，所以软件的编码十分简单。

1.5.5. 检错码和纠错码

数据在计算机系统中的生成、处理、存储和传输过程中都可能会发生错误。为减少和避免这种错误，除了提高硬件的可靠性外，在数据的编码上也应当提供检测和纠正的支持。即在代码中加入某种特征或规则，使得在发生了错误的时候，计算机能够发现错误并确定错误的位置，以便纠正出现的错误。在数据编码中，能够发现错误的编码称为检错码，能够纠正错误的代码称为纠错码。检错码和纠错码统称为数据校验码。数据校验码是能够检测或纠正代码中错误的信息编码，用于提高计算机系统的可靠性。

数据校验码可分为分组码和卷积码两类。分组码把信息序列分成以 k 个码元为单位的组，通过编码器将每组的 k 个信息位按一定规律产生 r 个冗余位〔校验码〕，输出长为 n=k+r 的一个码元组。在校验码中，符合校验规则的是“合法”代码，否则是存在错误的“非法”代码，代码的每一种组合构成一个码字(Code Word)。分组码中每一码元组的 r 个校验码元与本组的数据代码有关，而与别组的数据无关，一组内出现的错误只根据组内的代码进行纠错。另一类数据校验码为卷积码，它把所有的信息一起进行编码，使代码相关性更大，纠错能力更强。分组码可用（n，k）表示，n 表示码长，k 表示信息位数目。k 与 n 的比值称为校验码的编码效率。常用的数据校验分组码有奇偶校验码、海明校验码和循环冗余校验码(CRC)等。

奇偶校验码是一种最简单的分组检错码，它在每个分组的信息位中增加 1 个校验位代码，能够检测出代码中的奇数个位的错误，但不能纠正错误。常用于对存储器数据的检查或者传输数据的检查。偶校验码的构成规则是：所有信息位和单个校验位的模 2 加等于 0，即每个码字（包括校验位）中 1 的数目为偶数。奇校验码的构成规则是：所有信息位和单个校验位的模 2 加等于 1，即每个码字中 1 的数目为奇数，表示为

奇校验：$x_1 + x_2 + \cdots + x_k + x_{k+1} = 1 \quad \text{mod} 2$

偶校验：$x_1 + x_2 + \cdots + x_k + x_{k+1} = 0 \quad \text{mod2}$

其中 x_1，x_2，…，x_k 而为信息位，x_{k+1} 为附加的校验位。这里，模 2 加运算是在加法完成后对 2 取模，它等价于数字逻辑中的异或运算。根据数据位构成校验位的规则是：

奇校验：$x_{k+1} = x_1 + x_2 + \cdots + x_k + x_{k+1} + 1 \quad \text{mod2}$

偶校验：$x_{k+1} = x_1 + x_2 + \cdots + x_k \quad \text{mod2}$

在发生了单个错误时，代码就不再符合上述代码的规则，错误就能被检出来。这种校验码不能确定发生错误的确切位置，所以无法进行纠错。奇偶校验码可用于信息的传输或者信息存储中的编码，如在将信息从一个位置传输到另一个位置时对信息进行奇偶校验。在信息发送端，信息输入到一个奇偶校验位生成电路，该电路根据信息位产生校验位，信息位连同校验位一起送往接收位置，在信息接收位置，信息位和校验位都输入到奇偶校验电路中进行检查。

在信息编码中，两个合法代码对应位上编码不同的位数称为码距，又称海明距离。 在一种编码中各个码字间距离的最小值称为该代码的最小码距么最小码距是数据编码的一个重要特征。例如，代码 0000 与 0001 之间有 1 位不同，它们的码距为 1；代码 0001 与 0010 之间有 2 位不同，它们的码距为 2。

一般数据代码中允许代码有各种组合，代码之间的最小码距为 1。数据校验码的编码原理是在编码中引人一定的规则和冗余位，增加代码的最小码距，使得编码中出现一 个错误时就成为非法代码。奇偶校验码的原理是在每组代码中增加一个冗余位，使合法代码的最小码距由 1 增加到 2。合法编码中如果有一个位发生了错误，这个编码就将成为非法的代码，因而能检测出一组代码中的一个错误。增加的冗余位称为奇偶校验位。 如果奇偶校验码中出现了两个错误，它就变成另一个合法的代码，同样满足编码规则， 所以不能判断出是否存在错误。下面对奇偶校验码作进一步的研究，以引出校验码中的一些基本概念。

设一奇偶校验码

$$\omega = \{\omega_1, \omega_2, \cdots, \omega_n\} (n = k + 1)$$

如果校验位为，则

$$\omega_n = \omega_1 + \omega_2 + \cdots + \omega_{n-1}$$

这里的加法为模 2 加法，它不保留进位，其结果与异或运算相同。上式可表示为

$$\omega_1 + \omega_2 + \cdots + \omega_{n-1} + \omega_n = 0$$

这个方程称为偶校验的校验方程。在代码的存储或传输中，当写入或发送的码字为 ω，读取或接收到的代码为

$y=\{y_1,\ y_2,\cdots,\ y_n\}$

在错误源的干扰下，接收的代码等于原来的信息代码加上由错误源产生的干扰信号 $e=\{e_1,\ e_2,\cdots,\ e_n\}$（称为错误模型），则

$$y=\omega+e=\{\omega_1+e_1,\omega_2+e_2,\cdots,\omega_n+e_n\}$$

把接收码字 y 代入奇偶校验方程，可得

$$s=y_1+y_2+\cdots+y_n$$

如果发送的码字 ω 如能满足奇偶校验方程

$$\begin{aligned}s&=\omega_1+e_1+\omega_2+e_2+\cdots+\omega_n+e_n\\&=e_1+e_2+\cdots+e_n\end{aligned}$$

这里 s 称为校正子或伴随式(Syndrome)。如果没有错误，则 $e_1=e_2=\cdots=e_n=0$，s 也等于 0。当产生奇数个错误时，s=1 因此，奇偶校验方程能够检测出奇数个错误。

第一章 习题

1. 解释下列术语。

ALU 寄存器 CPU 运算器 ALU 外围设备 数据 指令

位 字 字节 字长 地址 存储器 主存储器 辅助存储器

存储器的访问 顺序访问存储器 总线 硬件 软件 兼容 透明

操作系统 汇编程序 汇编语言 编译程序 解释程序 系统软件 应用软件

指令流 数据流接口 虚拟机 处理机 处理器 ROM RAM

SRAM DRAM DSP MCU

2. 电子计算机一般分成哪些组成部分？为什么要分成这些组成部分？

3.运算器中可以有哪些寄存器？为什么？

4.存储器为什么要分为内存和外存？

5.什么是存储器的容量？什么是数据字？什么是指令字？

6.存储器中可存放大量数据，怎样从中找出指定的数据？

7.什么是存储器的读操作？什么是存储器的写操作？什么存储器的访问？

8.存储器中每一个字节都有一个地址，当我们要从存储器中读取一个 32 位的字， 也就是 4 个字节时是否需要向存储器提供 4 个地址呢？

9.什么是 CPU？什么是主机？输入/输出设备为什么称为外围设备？

10.当 CPU 中有许多存放数据的寄存器时，如何指定某一个数据寄存器？

11.硬件通常位于机箱内，为什么说它是外围设备？

12.软件与硬件之间有什么关系？

13.什么是计算机程序设计语言？为什么要有程序设计语言？

14.什么是虚拟机？什么是 Java 虚拟机？

15.高级语言有哪些特点？

16.嵌入式计算机有何特点？

17.解释下列术语。

原码　补码　反码　移码　阶码　尾数　基数

机器零　上溢　下溢　ASCII　Unicode　BCD　规格化数

海明距离　检错码　纠错码　海明码循环码　包装字

18.将以下十进制数据表示成二进制数，小数点后保留6位。

（1） 35　（2）67　（3）103　（4）52　（5）5.5　（6）120.125

19.将下列二进制数转换成十进制数。

（1） 10011101　（2）10110110　（3）10000111　(4) 00111000

20.将下列十进制数转换成二进制数，再转换成八进制数和十六进制数。

（1） 234　（2）1023　（3）131.5　（4）27/32

21.将100 1011102转换成八进制数和十六进制数。

22.将十进制数0.81转换成二进制数。

23.将167_8和$1C4_{16}$转换成十进制表示。

24.求以下编码的真值：

（1） $[x]_{原}=1010101$

（2）$[x]_{原}=0010101$

（3）$[x]_{补}=0011101$

（4）$[x]_{补}=1011101$

25.在定点整数计算机中，若寄存器的内容为80_{16}，当它分别代表原码、补码、反码和无符号数时，所对应的十进制数各为多少？

26.写出下列二进制数的原码、反码、补码和移码

⑴ 11010100

⑵ 0.1010000

⑶ -10101100

⑷ - 0.0110000

27.根据$[x]$补如何求$[-x]$补？

28.浮点数的阶码为什么通常采用移码？

29.什么是规格化数？如何判断一个数是否是规格化数？

30.对以下数据作规格化浮点数的编码，假定 1 位符号位，基数为 2，阶码 5 位， 采用移码，尾数 10 位，采用补码。

⑴ 10110_2

⑵ -0.00138_{10}

31.以 2 为基数，有 1 位符号位、4 位阶码和 8 位尾数的浮点数，阶码采用移码表示，求数值表示范围及可表示的数据个数。

32.以 16 为基数，有 1 位符号位、4 位阶码和 8 位二进制尾数的浮点数，阶码采用移码表示，求数值表示范围及可表示的数据个数。

33.以 R 为基数，有 1 位符号位、4 位阶码和讲位二进制尾数代码的浮点数，阶码采用移码表示，求数值表示范围及可表示的数据个数。

34.设浮点数的格式为：

符号位：b15；

阶码：b14～b8，采用补码表示；

尾数：b7～b0，与符号位一起采用规格化的补码表示，基数为 2。问：

⑴它能表示的数值范围是什么？

⑵它能表示的最接近于 0 的正数和负数分别是什么？

(3)它共能表示多少个数值？

请用十进制数 2 的幂次表示。

35.上题中，如果基数改为 16，尾数用原码表示，则结果怎样？

36.将正数+0.4 用二进制浮点数表示，阶码为 4 位，尾数为 7 位，指出该表示的相对误差。

37.某机字长 32 位，浮点表示时，阶码占 8 位，尾数占 24 位，各包含一位符号位，问：

(1)带符号定点小数的最大表示范围是多少？

(2)带符号定点整数的最大表示范围是多少？

(3)浮点表示时，最大正数是多少？

(4)浮点表示时，最大的负数是多少？

(5)浮点表示时，最小的规格化正数是多少？

38.将下列十六进制的单精度数代码转换成十进制数值表示。

⑴ 42E48000　　(2) 3F880000　　⑶ 00800000　　⑷ C7F00000

39.将下列数据用 IEEE 754 单精度浮点格式表示。

⑴-5　　⑵-6　　⑶-1.5　　⑷384　　(5) 1/16　　(6)-1/32

40.设计一个浮点数据格式，用尽量少的位数满足以下要求：

⑴数值范围为$-1.0\times10^{38}\sim-1.0\times10^{-38}$和$1.0\times10^{-38}\sim1.0\times10^{38}$。

(2)精度为表示 7 位十进制数据（相对精度）。

(3)用全 0 表示数据 0。

41.试将以下文字信息用 ASCII 代码表示：

(1)MS-DOS

(2)Serial Number is A679-3C10

42.二进制信息 10001111 代表什么？假定它是

(1)一个整数的补码。

(2)一个无符号整数。

(3)一个定点小数的补码。

(4)一个无符号整数的奇偶校验码，最右 1 位为校验位。

43.对于二进制机器数 1000 1111 1110 1111 1100 0000 0000 0000：

(1)表示一个补码整数，其十进制值是多少？

(2)表示一个无符号整数，其十进制值是多少？

(3)表示一个 IEEE 754 标准的单精度浮点数，其值是多少?

第2章. 中央处理器与指令系统

计算机的控制器与运算器一起构成计算机的中央处理器（CPU）。控制器是计算机 CPU 的关键部件之一，它与 CPU 的结构、指令的执行过程密切相关。本章首先介绍 CPU 的基本结构，然后介绍指令的执行过程，再介绍指令流水执行的概念，最后将介绍指令的概念和常用的指令寻址方式。

2.1. CPU 的基本概念

2.1.1. CPU 的基本功能

CPU 具有以下四个方面的基本功能：

（1）指令控制。即对程序运行的控制。程序由一个指令序列构成，这些指令在逻辑上的相互关系不能改变。指令之间的逻辑关系包括数据的依赖性和控制依赖性，即数据流的相关性和控制流的相关性。CPU 必须对指令的执行流程进行控制，保证指令序列执行结果的正确性。

（2）操作控制。对指令的各个操作步骤进行控制，即指令内操作步骤的控制。一条指令的功能一般需要几个操作步骤来实现，CPU 必须控制这些操作步骤的实施。操作控制包括对各种操作的时间控制，即对各种操作进行时间上的控制。因为各种操作信号在时间上有严格限制，所以必须控制各信号之间的相互关系。

（3）数据运算。即对数据进行算术运算、逻辑运算等各种运算，包括定点数、浮点数、字符数据等的运算。这是 CPU 的最基本的功能，由各种运算器的功能部件实现。

（4）异常处理和中断处理。对 CPU 内部出现的意外情况进行处理，如处理数据运算中的溢出等错误情况以及处理外围设备的服务请求等。

此外，CPU 还可具有存储管理、总线管理、电源功耗管理、Cache 及其管理等扩展功能。存储管理就是虚拟存储器的管理以及存储器的保护等。总线管理是对 CPU 所连接的系统总线的裁决、同步等。电源管理是为为了减少 CPU 的电源功率消耗以及减少 CPU 芯片的发热，对芯片内部的供电和工作频率进行配置和管理。

CPU 中主要包括控制器、运算器、寄存器等，此外 CPU 中还可包含 Cache、存储管理部件等。CPU 的上述功能除了数据运算外，主要由控制器完成。控制器的主要功能是控制指令执行的步骤和数据在各部件之间的流动，包括从内存中取指令、计算下一条指令在内存中的地址、对指令进行泽码、产生相应的操作控制信号等。在采用流水技术的 CPU 中，控制器还要对流水线的工作过程进行控制。

运算器接受控制器的命令进行运算操作。它是指令执行部件，又称功能部件。运算器的功能是执行所有的算术运算和所有的逻辑运算，包括进行比较测试。许多处理器中具有多个运算功能部件，这些功能部件能够并行地进行运算。

2.1.2. CPU 中的寄存器

在 CPU 中可以有多种寄存器，以存放各种信息。如在 CISC（复杂指令集）系统中一般有五种类型的寄存器：

（1）指令寄存器（IR，Instruction Register）。存放当前正在执行的指令，为指令译码器提供指令信息。指令寄存器对程序员透明，对 IR 的操作是隐含的。

（2）程序计数器（PC，Program Counter）。存放指令的地址，从存储器中取指令时根据 PC 的值进行。取完指令后又将 PC 更新为下一条指令的地址。

（3）数据寄存器（DR，Data Register）。存放操作数、运算结果和运算的中间结果，以减少访问存储器的次数，或者存放从存储器读取的数据以及写入存储器的数据。

（4）地址寄存器（AR，Address Register）。存放操作数的地址。在 RISC 处理器中一般设置通用寄存器，既可存放地址，又可存放数据。

（5）状态寄存器（SR，Status Register）。存储运算中的状态，作为控制程序的条件，如数据比较的结果和各种出错的情况等。在某些计算机的控制器中，将反映机器运行情况的状态代码集中在一起，构成程序状态字（PSW），存储在状态寄存器中。在 RISC（精简指令集）处理器中一般不设置定点数据的状态寄存器。

CPU 中的寄存器有些是程序员可见的，有些是内部的寄存器，程序员看不见。程序员可见的寄存器是程序员可以通过汇编语言访问的寄存器。上述五种寄存器都是程序员可见的。CPU 中的内部寄存器包含指令寄存器 IR 和一些存储器接口寄存器等。

2.1.3. 数据通路

在计算机执行指令的过程中，需要不断地在寄存器和 ALU 之间传递数据。通常把寄存器与 ALU 之间传递信息的线路称为数据通路。数据通路的结构构成了 CPU 的主体结构。数据通路中的数据传递操作在控制器的控制下进行。数据通路的建立一般有以下两种方法：

（1）用数据总线。在各寄存器以及 ALU 之间建立一条或者几条公共的数据总线，寄存器间的数据传输通过这些总线完成。一条总线可以连接多个部件，总线连接方式可以减少线路的数量。总线上可以有多个部件同时接收数据，但任一时刻只能有一个部件向同一条总线发送数据。因此，连接到总线上的寄存器或者其他部件需要进行输出端控制，以防止总线上的数据冲突。在总线结构中，可同时进行数据传输的数量取决于总线的数量。如果数据通路只用一条总线构成，则一次传输一个数据，称为单总线结构。

（2）用专用的通路。在各寄存器与 ALU 之间根据指令执行过程中的操作和数据流向来安排功能部件并建立相应的数据传输通路。每条数据传输线路都是专用的，不共享使用。这样建立的数据通路在数据传输和操作中可以做到互不相关，控制比较简单，各寄存器之间的数据传输可以并行进行，从而可以达到较高的性能，但在部件数量多的情况下需要建立的通路数量很多，使得结构较为复杂。

寄存器之间的数据传输除了上述数据通路外，还可通过算术逻辑单元进行。每个通用寄存器都有通往算术逻辑单元的通路和接收算术逻辑单元结果数据的通路。这样，算术逻辑运算部件也是寄存器间传输数据的一条通路。这种数据传输方式比较灵活，但算术逻辑单元在

进行传输操作时就不能进行运算操作．数据传输要占用算术逻辑单元，影响工作速度；因此一般作为上两种方法的补充。

2.1.4. 单总线数据通路

图 2-1 是一种最简单的单总线数据通路的 CPU 结构。在这个 CPU 结构中，包含一个指令寄存器 IR、一个指令地址计数器 PC、一个访存接口地址寄存器 MAR、一个访存数据接口寄存器 MDR、n 个通用寄存器 R0~Rn-1，和运算器 ALU。各部件的数据传递都要经过这条唯一的总线。从硬件结构上看，单总线结构是最简单的结构，实现成本低，但是由于总线的使用冲突会影响系统的性能。在单总线结构的数据通路中，为了向 ALU 提供运算的数据，还设置了 Y 寄存器和 Z 寄存器，因为单总线不能同时向 ALU 提供两个操作数并且传输运算结果。Y 寄存器用于存放一个操作数并向 ALU 的一个端口提供操作数据，向 ALU 提供数据需要分成两步，先将一个数据放入 Y 寄存器中。ALU 的结果输出也要分成两步，先将结果放入 Z 寄存器中，然后通过总线送到指定的通用寄存器或其他寄存器中。在单总线数据通路中，在 ALU 的另一个输入端口也可以设置一个数据寄存器。

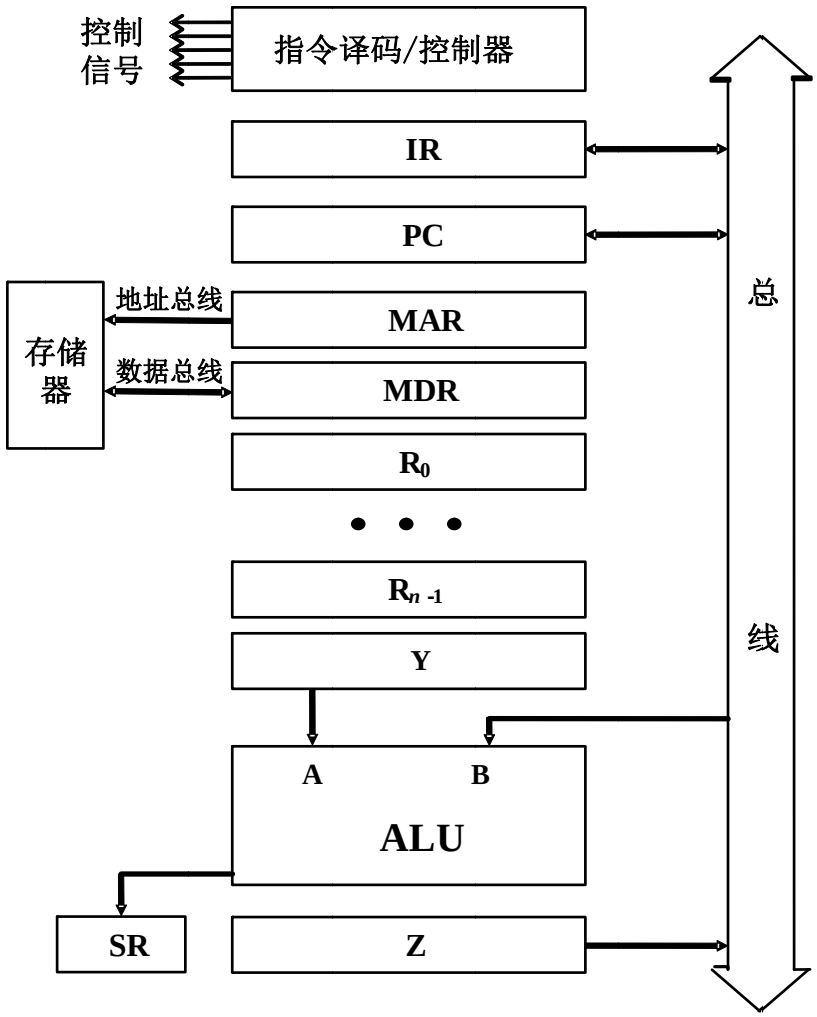


图 2-1　单总线 CPU 的结构

CPU 中的各个部件的操作一般都需要相应的控制信号，以控制操作的内容和时间。在单总线结构的 CPU 中，总线上每次只能传递一个数据，连接在总线上的每个部件都需要有输入控制信号和输出控制信号，以控制在每一个操作周期内由哪一个部件向总线发送数据，以及由哪一个部件接收数据。对于寄存器部件来说，输入控制用于控制总线上的数据输送给寄存器，寄存器的输出控制用于控制将寄存器中的数据输送到总线上。此外，PC 是一个具有计数功能的寄存器，需要有计数控制信号；ALU 有各种计算功能，需要有运算控制信号。

对于图 2-1 所示的结构，所需要的控制信号包括以下几种：

R1out： R1 的输出控制。

R1in：R1 的输入控制。

Yin：Y 寄存器的输入控制。

IRin：IR 的输入控制。

IRout：IR 的输出控制。

PCin：PC 的输入控制。

PCout：PC 的输出控制。

PC＋1：PC 计数更新控制。

ADD：ALU 加法操作控制。

SUB：ALU 减法操作控制。

在安排控制信号的时序时要注意在总线上所有部件中任何时刻只能有一个部件向总线输送数据，因此在连接到总线的所有部件的输出控制信号中，不能有两个同时为有效信号。当控制信号为高电平有效时，不能同时有两个输出控制信号为高电平。

单总线结构 CPU 是最简单的 CPU 结构，由于一次只能在总线上传输一个数据，所以性能较低。如果采用两条总线，则构成双总线结构的数据通路。双总线结构的数据通路略为复杂，它采用一条数据总线（DBUS）和一条指令总线（INS）连接数据通路和指令通路，可加快指令的执行速度。有关双总线结构的概念和运行原理这里不作详述。

2.1.5. 指令的执行和周期概念

计算机程序在运行之前装入到主存储器中。在程序的执行过程中，CPU 从主存中逐条取出这些指令，依次执行。在没有遇到分支指令（转移指令等）时程序中的指令是按顺序执行的。CPU 用 PC 寄存器保存指令的地址，当取出一条指令之后，PC 更新为下一个指令字的地址。存储器的访问操作是按字进行的，当指令的长度大于一个字时，上述过程就需要重复进行。这个步骤称为指令执行的取指阶段。取指阶段之后则进入执行阶段。首先分析指令，对指令进行译码，识别指令所要进行的操作，并产生相应的操作信号，如果参与操作的数据在存储器中，还需要形成操作数的地址并读取操作数据。执行指令时根据指令分析产生操作控制信号，完成指定的操作。最后将执行结果写回寄存器或存储器。然后再读取下一条指令，并分析和执行，如此循环。

从一条指令的启动到下一条指令的启动的时间间隔称为指令周期。指令周期中包含若干

个基本操作步骤，如访问存储器和运算等。每个基本操作的时间称为机器周期。指令中包含的机器周期数取决于指令的功能。机器周期是指令执行中每一个基本操作所需的时间，它代表了大多数指令操作步骤的时间。机器周期基本上是根据存储器的速度及 ALU 执行周期的基本时间确定的。一个机器周期可以包括若干个时钟周期。时钟周期是计算机时钟主频的周期。因为各种指令的操作功能不同，因此各指令的执行周期数也各不相同。早期的计算机中一个指令周期一般需要几个机器周期完成，一个机器周期需要几个时钟周期。近年的新型计算机中采用了硬件并行的技术以及简化的指令系统，使得机器周期一般等于一个时钟周期，平均指令周期可以等于甚至小于一个时钟周期。在本章后面的例子中，假设访存和运算等操作都可以在一个时钟周期内完成，因而机器周期等于时钟周期。

2.2. 指令周期

上面已经对指令的执行过程作了简单介绍，指令的具体执行过程取决于具体的指令类型以及 CPU 的数据通路结构。根据执行的过程，指令的类型可以分为运算指令、访存指令和控制指令。每一种指令的执行过程比较接近。下面主要针对总线型结构对这三种最常见的指令类型的执行过程作进一步分析。

2.2.1. 运算指令周期

在单总线的 CPU 结构中，为了使连接在总线上的各模块之间能够相互传递数据，必须控制各模块的输入 / 输出门。特别是输出控制，对总线上的每一条信号线，在整个总线上和任一时刻只能有一个模块驱动总线上的这个信号线。电路模块的输出控制采用三态门控制向总线的信号输出，输入控制则通过控制各模块的输入时钟。模块中数据输入 / 输出的控制可以用专门的总线驱动器件或者集成在模块的其他器件中，例如在寄存器中可集成输入控制和输出控制的功能。CPU 内部的总线数据传输过程由系统的控制器完成，由控制器统一向各模块发出输入 / 输出控制信号。图 2-2 是总线操作的例子，其中总线上的三次数据传输分别是：R1→Y，R2+Y→Z，Z→R3，分别发生在 T1、T2 和 T3 周期。这里还假设输出控制信号为电平控制信号，即数据输出在整个周期内有效，而输入控制信号为后边沿触发，即在输入控制信号的下降边沿将总线上的数据输入到功能模块中。

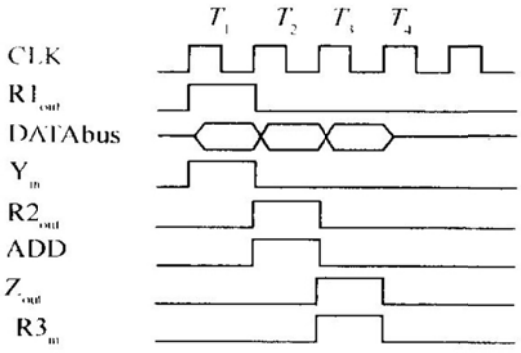


图 2-2 总线操作时序图

在单总线的 CPU 结构中，存储器是 CPU 外部的部件，取指令的操作需要通过接口寄存器 MAR 和 MDR。CPU 中的 ALU 是一个组合逻辑电路，在进行加法等运算时，两个源操作数必须同时提供给 ALU 的输入端。在将运算结果写入寄存器之前，源操作数输入端的数据必须保持不变，所以运算指令需要用到 Y 和 Z 这两个内部寄存器。例如在单总线结构中，

将 R1 和 R2 中的数据相加，结果送入 R3 的指令“ADD R3，R1，R2”的执行过程是：首先将 PC 的值通过总线送到地址寄存器 MAR，进行读指令操作，同时将 PC＋1 送到 PC（这里假设指令的长度为一个字，PC 的值为字地址而不是字节地址），使 PC 的内容为下一条指令的地址，指令读出后通过存储器的数据总线 DBUS 送到数据寄存器 MDR，再通过 CPU 内部的总线送到指令寄存器 IR。然后对 IR 中的指令进行译码，译码后进入执行阶段。在执行阶段中，将通用寄存器 R1 中的数据送到 Y 寄存器，将 R2 中的数据送到 ALU，与 Y 中的数据相加，结果先放入 Z 寄存器，最后将 Z 寄存器中的结果写入 R3。这些步骤可表示为：

PC→MAR，读存储器。

PC＋1→PC。

DBUS→MDR。

MDR→IR。

R1→Y。

R2+ Y→Z。

Z→R3。

上述前 4 个步骤是取指阶段，PC 的加 1 操作通过 ALU 进行。PC 寄存器也可以是一个计数器实现。如果 PC 寄存器具有计数功能，则可减少对总线的占用。

2.2.2. 访存指令周期

访存指令分为读操作和写操作两种。

在单总线的 CPU 结构中，执行读存储器数据到寄存器的指令“LOAD R1，mem”的过程是：

PC→MAR，读存储器。

PC＋1→PC。

DBUS→MDR。

MDR→IR。

IR（地址段）→MAR，读存储器。

DBUS→MDR。

MDR→R1。

前面 4 步与上述运算指令的一样，第 5 步将指令中的地址码字段送到 MAR，进行读数据的操作，从存储器中读出的数据先送入 MDR，然后再通过总线送到 R1 中。

写操作与读操作的不同之处在于：写操作时，CPU 不但要向存储器提供地址，还要提供数据。地址通过 MAR 提供，数据通过 MDR 提供，在数据和地址都提供之后发出写操作控制信号。例如，写存储器指令“STORE R1，mem”的执行过程是：

PC→MAR，读存储器。

PC＋1→PC。

DBUS→MDR。

MDR→IR。

IR（地址段）→MAR。

R1→MDR，写存储器。

2.2.3. 控制指令周期

转移指令是最为常见的程序控制指令。转移指令分为条件转移指令和无条件转移指令。在计算机中，条件转移指令有三种常用的安排方法。

（1）条件码方法，就是由 ALU 操作设置特定的位作为条件码，这个条件码存放在状态寄存器中。每条指令执行完成时都将它的结果条件写入状态寄存器。这种方法的优点是有些状态位可以自由地设置，从而可构成各种判断条件。其缺点是条件码是一种额外状态，是指令产生的副作用，大多数情况下状态位不被利用。由于条件码是从一条指令传递给条件转移指令，限制了指令的执行顺序。

（2）条件寄存器方法，就是将条件码放入通用寄存器中，测试带比较结果的任意通用寄存器。只有测试指令的执行结果才写入指令指定的通用寄存器，其他指令不写结果条件。这种方法的优点是简单，避免指令隐含的副作用，也避免了不必要的状态写入，缺点是占用了一个通用寄存器。

（3）比较与转移方法，即比较操作是条件转移指令的一部分功能。这种方法的优点是不需要用寄存器存放条件码；条件转移只要一条指令完成，而不是两条；这样可以不限制指令的顺序，指令的顺序可以根据需要进行调整，以提高执行速度。这种方法在 RISC 机中广泛采用。

转移指令通过修改 PC 的内容来改变程序的流程。修改 PC 是转移指令的主要操作。一条转移指令的操作过程是：

（1）取指令。将程序计数器 PC 的内容作为地址访问指令存储器，并将 PC 的内容加上指令的字节数，访问到的内容传送到指令寄存器 IR。

（2）指令译码。对指令寄存器中的操作码进行译码，识别指令操作类型。对于条件转移指令，则进行条件判断，这时需要读取寄存器，ALU 可能需要进行比较操作。

（3）计算下一条指令地址。根据指令的寻址方式计算下一条指令的地址，并将计算结果送入 PC。在简单情况下，下一条指令采用直接寻址方式，直接将指令寄存器中的地址码部分送到程序计数器 PC。

对于单总线结构的计算机，在执行相对转移指令 JMP offs 时的控制操作步骤是：在将指令读入 IR 后将 PC 的值送入 Y 寄存器，将 IR 中的地址偏移量部分送到 ALU 与 Y 寄存器的值相加，形成的转移目标地址先送入 Z 寄存器，再送入 PC 寄存器。因为转移目标地址与 PC 的值相关，转移目标地址与当前指令的存储位置相关，所以这种转移操作是相对转移。相应地，如果转移目标地址与 PC 的值无关，这种转移操作就是绝对转移。上述这条指令的执行过程描述如下所示。

PC→MAR，读存储器。

PC＋1→PC。

DBUS→MDR。

MDR→IR。

PC→Y

Y+IR（地址段）Z

Z→PC

如果 JMP 指令是一条绝对转移指令，则只需直接将指令中给出的存储器地址送入 PC。

2.3. 指令的流水线技术

2.3.1. 指令的流水执行

流水技术在计算机中用于提高指令的执行速度和数据运算的速度。计算机的流水工作方式是将一个计算任务细分成若干个子任务，每个子任务由专门的部件（流水段）处理，这些部件构成一条流水线。计算机流水线中的计算任务可以是一个算术逻辑运算操作，也可以是一条指令的执行。指令级流水线则是把一条指令的执行过程分成多个子过程，由各个部件进行轮流处理后完成执行过程。这样，不必等到上一条指令的完成就可以开始下一条指令的执行。指令级流水线在高性能的微处理器中被普遍采用。

指令流水线由一系列串联的流水段组成，每个流水段完成指令执行的一个操作步骤。各个流水段之间设有缓冲寄存器（称为流水寄存器），以暂时保存上一流水段对指令处理的结果。流水线中每个流水段构成流水线的一级。在专用通路结构的 CPU 中，通常将指令的执行分为取指、译码、执行、访存及写回 5 个流水段进行流水处理。指令的流水执行不能缩短一条指令的指令时间，但是通过指令间的重叠执行方式可以提高指令执行的速率或吞吐率。

指令流水线的吞吐率（throughput）是衡量指令流水线的一个重要指标，指单位时间内流水线能执行的指令数量。流水线的吞吐率与流水的节拍时间有关。指令流水节拍是指令从一个流水段进入下一个流水段的间隔时间，又称为流水周期。指令流水线通常要求每个流水段同步地进行移动，在一个统一的时钟控制下，指令从一个流水段流向下一个流水段，也就是每一级流水段同时将指令执行的中间结果放在流水寄存器中，传递给下一个流水段。流水周期的确定也可以有多种方式。最简单的方法是将流水周期定为各流水段处理时间的最大值，加上流水寄存器的延迟时间。

指令的执行采用流水方式后，CPU 中各部件的利用率提高了。原来在一条指令的执行过程中，取指部件读取一条指令后就暂停工作，由译码部件对指令进行译码，然后进行其他的操作。每个部件只在指令执行过程的一两个阶段进行操作，其他阶段不进行操作。在指令流水线中，取指部件在每个流水周期中都要读取一条指令，译码部件在每个流水周期都完成一条指令的译码，ALU 可能在每个周期都要进行一次数据运算。每个部件都不停地进行操作，从而提高了部件的利用率和指令执行的速率。

流水线的工作情况可以用一个时空图表示。图 2-3 是一个 5 级线性流水线在执行 8 个流

水任务时的时空图，该流水线采用固定的流水周期。图中，横坐标表示时间，以流水周期为单位，纵坐标为流水段编号，表示不同的流水部件，流水线中的一个子任务用一个小方块表示，其中的数字表示任务编号（在指令流水线中就是指令编号）。在指令流水线中，从第一条指令进入流水线到离开流水线的时间是流水线的建立时间，从最后一条指令进入流水线到离开流水线的时间是流水线的排空时间。

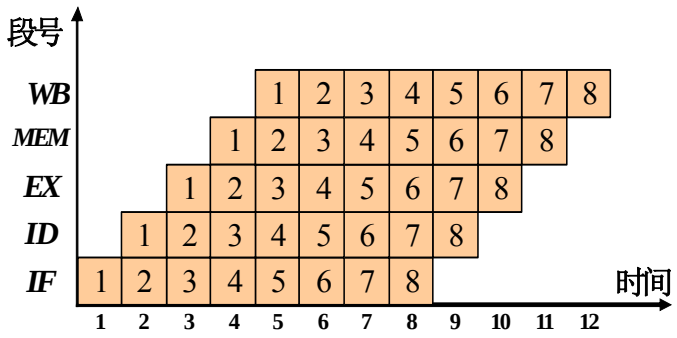


图 2-3　流水线的时空图例子

从时空图上可见，对进入流水线的每一条指令，其执行时间都没有缩短。指令流水线通过指令之间的重叠来提高吞吐率。在流水线中，当任务饱满时，任务源源不断地输入流水线，不论有多少个流水段，每隔一个流水周期都能输出一个任务，从而在宏观上提高了处理的速率。

上述流水线时空图是根据流水段来画的。这种时空图便于分析流水线硬件资源的使用情况。指令流水线的时空图通常根据指令序列来画，以便于分析指令之间的关系。这种时空图的纵坐标是代表指令序列的，用从上到下的箭头表示。时空图中每一行的小方格代表一条指令的流水阶段，每个阶段的名称在方格中的文字表示。在指令流水线中，数据相关性对指令流水执行影响的时空图如图 2-4 所示。其中，指令流水线分为取指（IF，Instruction Fetch）、译码（ID，Instruction Decode）、运算执行（EX，Execute）、访存（MEM，Memory）和写回（WB，Write Back）5 个流水段。

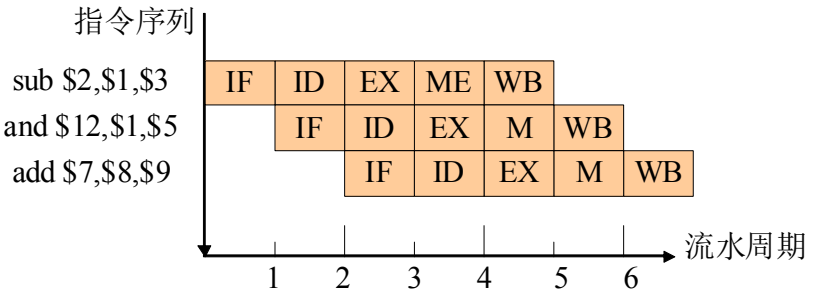


图 2-4 指令流水线的时空图

指令的流水执行方式是一种时间重叠的并行执行方式。在指令流水线中，可以在每个时钟周期就开始一条指令的执行过程，从而可以在每个时钟周期完成一条指令的执行。多条指令在流水线中重叠地执行，就可以大大加快指令的执行速度，提高计算机的性能。要使计算机的指令流水线具有良好的性能，必须设法使流水线能畅通流动。但是，在指令的并行处理

方式中会出现三种相关性，使得指令的并行执行性能受到限制。这三种相关是资源相关、数据相关和控制相关。

2.3.2. 指令流水线的相关性

1．资源相关

多条指令在同一段时间内需要使用同一个流水级，这种现象称为资源相关。这种相关性是因为硬件资源不够造成的，与硬件结构有关，所以又称结构相关。资源相关使得指令不能同时执行。例如在上述指令流水线中，如果数据和指令存放在同一个存储器中，而且只有一个访问接口，这样便会发生这两条指令需要同时访问存储器的情况。或者两条指令同时需要 ALU 进行地址或者数据的计算时，也发生了资源相关。解决资源相关的主要方法是增加资源，例如增加运算部件的数量、增加存储器模块。

2．数据相关

数据相关是流水线中指令之间的数据依赖关系，它使得相关的指令不能并行执行。例如，一条指令所需的操作数是另一条指令的运算结果时．这两条指令就存在数据相关。数据相关意味着指令之间存在数据读写的约束关系，这种约束关系要影响指令的并行执行。根据指令间对数据读写的先后次序关系，一般可将数据相关分为写后读（RAW）、读后写（WAR）和写后写（WAW）三种类型。写后读相关是上一条指令的输出数据与下一条指令的输入数据之间的相关；读后写相关是上一条指令的输入数据与下一条指令的输出数据之间的相关；写后写相关是两条指令的输出数据之间的相关。这些相关性使得数据的读写操作必须按顺序进行，如果改变了数据的读写顺序，就会产生错误结果。指令对数据的读操作之间的顺序不影响指令的正确执行，如果计算机中允许两条指令同时读同一个数据，就不存在读后读的相关性。在流水线中，数据相关可能影响指令的流水执行，也可能不影响，因此又称为险象（hazard）。

3．控制相关

控制相关是指令序列的选择关系，使得后继指令的选择由前面指令的执行结果来确定。计算机的程序是一个有分支的流程，流程中的分支导致了流水线的控制相关。在指令流水线中，控制相关主要由程序流控制指令引起。如果程序的流程中要求某一条指令进行控制转移，则在这条指令完成之前，无法确定哪一条指令是后继指令，使得后继指令不能进入流水线。

当流水线中存在相关性时，指令的流水执行就会受到影响。例如，在存在 RAW 相关时，上一条指令的结果需要传递给下一条指令。通常，上一条指令将结果写入一个寄存器，下一条指令从这个寄存器中读取数据。但是在流水线中，上一条指令可能在执行过程的第五个周期写回结果（如 load 指令），而下一条指令通常在第二个周期读取数据。这样，在下一条指令读取数据时，上一条指令还没有将数据写入寄存器。

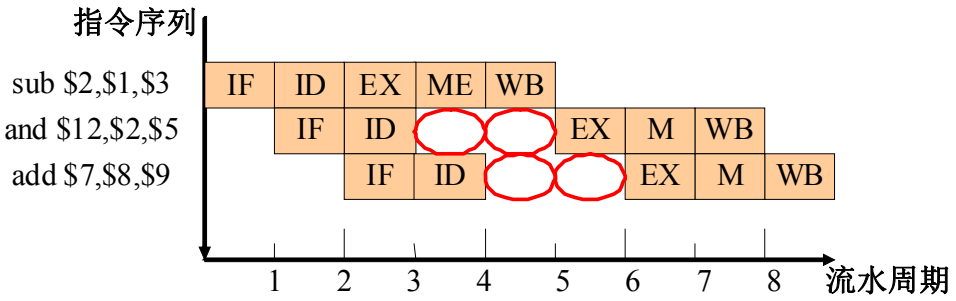
2.3.3. 解决相关性问题的方法

在指令流水线中，为解决相关性问题，需要停顿后继指令的流水执行，直至相关冲突能够避免。为此，首先需要硬件能够检测到指令间的这种相关性。指令间的数据相关性可以通过比较指令使用的寄存器号来实现。指令间相关性的检测也可以由编译程序来实现，编译程序可以比较各指令适用的寄存器，分析这些相关性是否影响指令的流水，避免将相关性的指

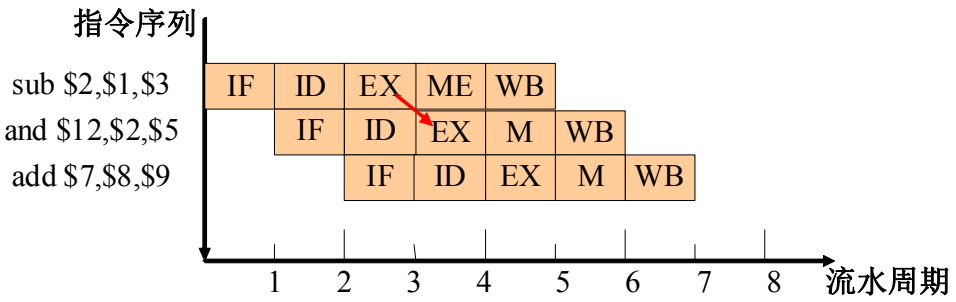
令放在一起，同时出现在流水线中。

相关性的检测也可以由编译程序完成。编译程序对生成的指令进行分析，在发现两条指令之间存在相关时，可以将其隔开，使其不同时出现在流水线中执行。为了将相关的指令隔开，可以改变指令的顺序，将其他指令移动到相关指令之间，或者在相关指令之间插入空操作指令（NOP）。

显然，流水线的停顿将使流水线的性能受到损失。所以，在计算机的设计中需要采取各种硬件和软件的措施来解决相关性问题，以避免流水线的停顿。在上述专用通路结构的指令流水线中，通过设置较多的数据传输线路来避免流水执行时的冲突现象，可避免一些资源的冲突。例如，可以在 ALU 的输出端到它的输入端之间设置一条数据线路，使得在 RAW 相关时，上一条指令的结果能够直接送到 ALU 的输入端，使得下一条指令能够及时得到上一条指令的结果，而不用到寄存器中去读取，从而可减少或避免了流水线的停顿。这种传递数据结果的快速通路称为相关专用通路（forwarding path），或称旁路通道 （bypassing path）。在数据相关时，没有相关专用通路和有相关专用通路时的三条指令流水的时空图例子如图 2-5 所示，图中的箭头代表指令间数据通过相关专用通道的传递。在图中的三条指令中，第一条指令的结果（$2）被第二条指令读取，所以存在写后读的数据相关胜，使得第二条指令产生停顿。第三条指令与前两条指令没有数据相关性，也停顿了两拍，形成有序执行的流水线；第三条指令也可以不停顿，形成无序执行的指令流水线。采用相关专用通路后，流水线中消除了停顿。



（a）没有相关专用通路



（b）有相关专用通路

图 2-5　指令流水线的时空图

控制相关也使得流水线中不能正常流水。因为当译码器发现一条转移指令时，下一条指令已经读取并进入流水线。如果转移指令要求从另一个地址读取下一条指令，那么已经读取的下一条指令就要作废。解决控制相关的方法是提前执行转移指令，或者对转移指令的转移

情况进行预测，并根据预测读取下一条指令，从而减少指令读取后被作废的概率。这种预测称为分支预测，它包括对转移指令是否会转移进行预测，已经对转移目标地址进行预测。为了进行这种预测，通常需要记录转移指令过去的行为历史。分支预测电路是一个状态机，记录转移指令过去转移状态以及目标地址的缓存称为分支目标缓存。

在流水线结构的 CPU 中，控制器产生的控制信号需要同时控制多条指令在不同执行阶段的操作。一种比较简单的控制方法是，控制器根据指令操作码同时产生该指令在后继各个阶段所需的控制信号，这些控制信号与该指令一起流水。不同指令流水阶段的控制信号构成不同的控制字段。在指令流到某一阶段时，取出该阶段的控制信号字段，对该阶段的操作进行控制。为此，在指令流水线的流水寄存器中需要增加控制信号的缓存。

由于指令的流水执行存在相关件和控制上的复杂性，不是任何指令系统都适合于流水执行。指令的流水执行要求各种指令的执行过程是相同的或者相似的，要求指令的执行过程中的步骤是固定的，要求指令执行中的各个步骤的时间是相同或者相近的。为了使指令间的相关性对流水的影响不致太大，指令的执行步骤不应太多。因为当指令的执行步骤很多时，流水线就很长，流水线中重叠执行的指令数量就很多，使得相关性很多，从而会发生很多停顿，而且需要复杂的判断和控制。为了使得指令执行中的各个步骤的时间是相同或者相近，要求指令的格式简单，以便于指令的读取和译码。

2.4. 指令格式和指令编码

每一台计算机的都有自己的一套指令，这些指令构成计算机的指令系统。指令系统是指计算机所具有的各种指令的集合，它反映了计算机硬件具有的基本功能。指令系统是机器语言和汇编语言程序员所看到的计算机的主要特性，也是系统程序员看到的计算机的特性。

计算机可直接执行的程序是由一系列的机器指令构成的。机器指令是计算机硬件能够识别并直接执行的操作命令。每条指令通常描述一个基本的操作，如进行一次加法运算或者乘法运算、到存储器中读取一个数据等。一台计算机系统中各种不同指令的集合称为这台计算机的指令集（instruction set），或称指令系统。指令系统反映了计算机的功能特征，也从一定程度上反映出计算机的结构特征。计算机是通过执行指令来完成其功能的。在计算机的指令中，为了指明它进行什么样的操作、操作数的来源、操作结果的去向，一般需要在指令中包含以下信息：

（1）操作的类型。说明操作的内容和功能，一般在计算机中可以有几十种以上不同内容的操作，如加法操作、访存操作、输入／输出操作等。每条指令都必须指定其操作的类型。

（2）操作数的存储位置。也称为操作数的地址，它说明参加运算的数据存储在什么地方。操作数的存储位置可以是寄存器、存储器单元，或者直接安排在指令中。在有些指令中，操作数的存储位置是默认的，或者说是隐含的，存储在一个固定的地方。

（3）操作结果的存储位置。它说明将运算结果存储存什么地方，如寄存器号或存储器单元的地址。在有此指令中，操作结果的存储位置也默认的。

（4）下一条指令的地址信息。它说明到哪里去取下一条指令。在大多数情况下，下一条指令的地址是默认的，即紧接着当前指令的存储位置。

指令中还可以包含条件信息，表示执行该指令操作必须具备的条件。为了表示不同的信息，指令中一般用不同的代码字段来表示。这种代码字段的划分和定义就是指令的编码。指

令的编码将指令分成操作码和各操作数地址码等几个字段，每个字段分别给出上述某个信息。计算机指令必须有一定的编码格式。指令编码的格式称为指令格式。

指定操作类型的代码称为操作码，指明操作数存储位置的字段称为操作数的地址码。操作数地址码有时可以直接就是参加操作的数据，但操作数通常存储在寄存器或存储器中，因此操作数地址码通常是寄存器号或者存储器的地址。操作结果的存储位置可以在指令中专门用一个字段指出，也可以确定为某个源操作数的存储位置。

下一条指令在多数情况下是当前指令存储位置的后面，也就是下一条指令的地址是当前指令地址加上当前指令的长度。这时可用一个程序计数器（Program counter，PC）指定下一条指令的地址，而不需要在指令中用一个字段指明。在某些情况下，程序需要转移到其他指令的位置，下一条指令就不是在当前指令存储位置的后面，这时可以专门安排一条指令，指出下一条指令的地址。这样的指令就是转移指令。它修改 PC 的内容，使其变成转移目标指令的地址。

一条指令的编码格式通常由操作码确定，包括指令的长度、字段数量和各个字段的长度，所以操作码是必不可少的. 通常是指令的第一个字段。指令格式的确定与计算机的字长、存储器的容量及指令功能都有很大的关系。每一种计算机都具有多种指令格式。图 2-6 是一种指令格式的例子，其中包含一个操作码和三个地址码。

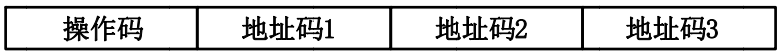


图 2-6 指令格式例子

2.4.1. 操作码

指令的操作码指定了指令的操作类型，也可以包含操作数的数据类型等信息。操作码可以是一个固定长度的代码，也可以是可变长度的代码。固定长度的操作码放在一个连续的字段中，所有的指令操作码长度相同，否则就是可变长度的操作码。固定长度操作码的编码方法十分简单，只要对不同的指令确定一个不同的二进制编码就可以了。固定长度操作码的编码也便于指令的译码，长度为 n 位的操作码可代表 2n 种不同的指令，但不能超过，因此不便于指令系统中增加新的指令。

可变长度的操作码便于增加新的指令，通过增加指令中操作码的长度，可以形成新的操作码编码。从指令的扩展性来看，希望操作码的长度可变。指令的扩展是在系统化计算机产品中，后继的计算机产品为了软件的兼容性需要保留先前计算机的指令及其编码格式，并且增加一些扩展指令，因为后继产品为了增加新的功能需要增加新的指令。为了使操作码具有可扩展性，原有的操作码编码不能全部用完，必须保留一部分代码作为增加新的操作码的前缀。如果增加了操作码的长度，就可以多增加一些操作码的编码；如果不增加操作码的长度，则只能使用保留的代码。

2.4.2. 地址码

指令中指定操作数存储位置的字段称为地址码。地址码中可包含存储器地址，也可包含寄存器号。指令中可以有一个、两个或者三个操作数，也可以没有操作数。根据一条指令中地址码的数量，可将指令分为零地址指令、一地址指令、二地址指令和三地址指令。四个地

址码以上的指令很少被采用。

零地址指令中只有操作码，而没有地址码。这种指令有两种情况，一种情况是无需操作数，如空操作指令 NOP（no operation）；另一种情况是操作数的存储位置为默认的，或称隐含的，如操作数在累加器中。累加器是一种存放数据的寄存器，通常同时连接在运算部件的输出端和输入端，便于进行累加运算。在运算部件中，一般都采用这种寄存器存放操作数和操作结果。

一地址指令的指令编码中有一个地址码。指令中只给出一个操作数的存储位置，这种指令可能是单操作数的运算指令或其他只需一个数据的指令，也可能是双操作数的运算指令，另一个操作数存储位置隐含为累加器，或者是一个隐含的常数。例如，INC（Increment）指令表示将一个操作数加 1，它只需要一个地址码。

二地址指令中包含参加运算的两个操作数的存储位置，运算结果通常存储在其中的一个地址中。大多数算术运算和逻辑运算都可以用二地址指令表示。这种指令的运算结果可以写入第一个地址码指定的存储位置，也可以写入第二个地址码指定的存储位置，不同的计算机有不同的规定。二地址指令也可以是其他需要两个数据存储位置的指令，如访存指令，它需要一个地址码指出数据的存放位置，另一个地址码指出存储器的地址。

三地址指令通常是运算指令，其中包含两个操作数的地址码和一个结果的地址码。与二地址指令表示的运算相比，它可以把运算结果写入其他的存储位置。三地址指令使得在操作之后源操作数不被覆盖，但如果三个地址都是存储器地址的话，这种指令将变得很长。通常情况是，指令中给出三个寄存器号，其中第一个寄存器号指定用于写入运算结果的寄存器，其他两个寄存器号指定提供运算的数据的寄存器。

对于一个加法运算 z←x＋y，其中 x、y 和 z 都是变量，存储在内存中。我们分别用 A、B、C 表示它们在内存中的存储位置。如果要用一条指令来完成这一运算操作，我们需要用一条三地址指令。指令中的 3 个地址码字段包含 3 个操作数的地址，操作码字段则表示进行加法操作。这样一个加法指令可表示为

ADD A，B，C

在三地址指令中，如果 3 个操作数字段都直接包含操作数的地址时，指令的长度将大于存储器地址长度的 3 倍。我们可以用比较短的指令来实现上述运算。如果在指令中只包含 x 和 y 的地址，那么运算结果存放到什么地址就需要进行约定。这样，在操作之后覆盖了一个源操作数。指令完成的操作是 x←x＋y 或者 y←x＋y，取决于不同的计算机，这里我们假定后一种情况。这样的指令可表示为

ADD A，B

为了防止源操作数的丢失，可以将其复制到另一个存储位置。复制数据的指令在一些计算机中用 MOVE 指令。即用“MOVE B，C”指令先将 B 中的数据复制到 C 中，然后将 A 和 C 中的数据相加之后送到 C。这样就需要两条指令：

MOVE B，C

ADD A，C

实现上述运算的另一种可能的方法是使用一地址指令。由于加法操作需要两个源操作数，因此必须隐含地假定其中一个源操作数和结果的存储位置，通常的做法是用一个累加器完成

这一功能。指令“ADD A”表示将存储位置 A 的数据与累加器中的数据相加，结果存放到累加器中。为了用这样的指令实现上述加法操作，需要引入两条新的一地址指令：LOAD 指令和 STORE 指令。LOAD 指令将指定存储地址中的数据装入累加器，STORE 指令将累加器中的数据存储到指定地址处。这样上述加法指令就可以用三条指令实现：

LOAD A

ADD B

STORE C

许多计算机中都有几个可作为累加器用的寄存器。当计算机中有多个累加器时，这种累加器被称为通用寄存器。在这种计算机的指令中，需要用地址码指定选择哪一个通用寄存器参加操作。

指令地址码除了可以指定寄存器号及存储器的地址外，也可以直接存放操作数。这种操作数又称为立即数，它直接包含在指令中，可以从指令中立即获得。

在指令格式中，操作数地址码长度可在很宽的范围内变化，而且可以有多种寻址方式的表示方法，所以只要适当安排指令的寻址方式就可与变长操作码很好地配合构成优化的指令代码

2.4.3. 指令字

一个指令字中包含二进制代码的位数，称为指令字长度。指令字长度应当与计算机的数据字长相匹配，以简化指令访存操作。一般指令的长度定为计算机字长的整数倍，至少是字节的整数倍。指令的长度可以是固定的，即一台计算机中所有的指令长度都相等；也可以是可变的，即不同的指令有不同的长度。

指令长度等于机器字长的指令称为单字长指令，指令长度等于两个机器字长的指令称为双字长指令。三字长以上的指令很少采用。单字长指令只需要一次访存就可以读取，使用双字长指令乃至多字长指令的目的在于提供足够的操作数地址信息，从而可增强指令的功能。固定长度指令的特点是指令的读取速度快。一般这种指令编码方式也较简单，但是有时为了凑满一定的长度，会使指令代码的效率降低。可变长度指令则可根据操作码和地址码的长度要求，灵活确定指令的长度，可构成功能较强的指令。但这种指令读取的速度比较慢，读指令时需要判断指令的实际长度。而指令的长度又要根据指令编码的内容来判断。

固定长度指令结构比较简单，可简化指令的译码。特别是单字长的等长指令，可加快取指令的速度以及指令的执行速度。

2.5. 操作数的存储及其寻址方式

存储器中既存储指令，又存储操作数据。在存储器中寻找指令或数据的方法有多种，如按地址寻找、按内容寻找、按顺序寻找等。在绝大多数计算机中都采用按地址寻找的方式，指令或数据的寻找问题变成了构成其地址的问题。在按地址寻找存储内容的计算机中，对指令的地址码进行编码，以形成操作数，操作数在寄存器或存储器中地址的方式称为寻址方式。

2.5.1. 操作数的类型和存储方式

存储器中既存储指令，又存储操作数据。操作数有多种类型，数据在存储器中又有不同的存储方式。

1. 数据的类型

指令可对不同类型的操作数进行操作。指令中表示操作数的类型有数值型、字符型等。数值型操作数可以是整型数或者浮点数。整型数可以是一个字节的数据、一个字的数据、长字数据或者 4 倍字数据。无符号数直接用二进制表示，带符号的数据通常采用补码编码。浮点数分为单精度数据和双精度数据，单精度浮点数是 4 字节长的数，双精度浮点数是 8 字节长的数。浮点数的表示大都选择 IEEE 754 标准编码。字符型数据一般采用 ASCII 码。

2. 操作数的存储方式

当一个数据元素的位数超过一个字节或者一个字的宽度时，这个数据就需要存储在相邻的多个存储位置。当 CPU 访问存储器中的一个字时，给定的地址是一个字节地址，实际上访问的是从这个地址起始的多个字节。例如，将一个 32 位的数据存放在地址为 100 的存储单元中，这个数据实际上存放在地址为 100、101、102、l03 的这四个单元中。每个单元中存放一个字节。

在多字节的数据中，按照高位字节和低位字节数据在存储器中的存储顺序，可分为大数端和小数端两种数据存储方式。将最低字节存储在最小地址位置的存储方式称为小数端方式；将最低字节存储在最大地址位置的存储方式称为大数端方式。图 2-7 所示是数据 1000000（即 000F424016）在存储器中存储方式的示意图。图中假定数据的存储地址是 100，数据字中各个字节的存储地址分别是 100、101、102、103。

图 2-7 多字节数据的两种存储方式

采用大数端方式的优点是：

（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，在对大数端字符串进行比较时，可以利用整型数的比较指令从高位到低位进行比较，可以同时比较多个字符。

（2）十进制数以及字符串的显示方便。在程序调试过程中需要显示内存中的数据内容，显示时按地址顺序进行。在大数端方式下，所有的数值型数据都可直接按从左到右顺序依次显示出数据的高位到低位，数据容易读取而不会产生混淆。

（3）顺序一致性。大数端计算机采用相同的顺序存储其整型数和字符串。

采用小数端方式的优点是：

（1）整型数据地址转换方便。大数端方式在将存储存存储器中的 32 位整数地址转换成 16 位整数地址时，由于最低字节存储地址的改变，必须作加法运算来调整数据的地址。在小数端方式下则不需要做这种调整。

（2）适合于超长数据的算术运算。超长数据运算的特点是数据结果的长度未知。对小数端方式存储的数据进行特别高精度的算术运算时，尽管不知道结果的位数，但可以先计算低位的数据，并将存储在内存中，然后计算高位数据，不需要根据先确定数据的位数来确定最高位的存储位值。

早期的计算机厂商形成了大数端和小数端两种方式。目前，许多新型计算机系统结构都同时支持大数端和小数端数据存储方式。

3. 数据对齐方式

所谓数据对齐的存储方式，是指数据的逻辑存储位置与物理访问位置的对齐方式。数据的物理访问方式根据存储器的连接结构有 8 位、16 位、32 位和 64 位等。我们已经知道，存储器的访问是按字地址进行的，CPU 提供的字节地址中的低位部分将不起作用。

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，它实际占据的存储器位置是地址为 4n、 4n＋1、 4n＋2 和 4n＋3（n 为自然数）。同样，如果字长是 16 位，字对齐方式要求一个数据字的存储地址是偶数。在按字对齐方式下，一个数据字的存储位置正好是存储器物理结构中一个字的位置，使得这个字的数据能够一次访问操作读取或者写入。如果这个数据字不按对齐方式存储时，其存储器地址就出现如 4n－1、 4n、 4n＋1 和 4n＋2 的情况。这样的数据在 32 位的存储器中也需要分两次读取或者写入。对齐的数据存储方式有利于简化访存接口并加快访存速度。在某些计算机中，规定数据的存储必须按字对齐的方式进行，为此有时需要跳开一些存储位置。

图 2-8 中是一个字的两种存储方式的例子，存储区域的有色部分表示存储的一个数据字。

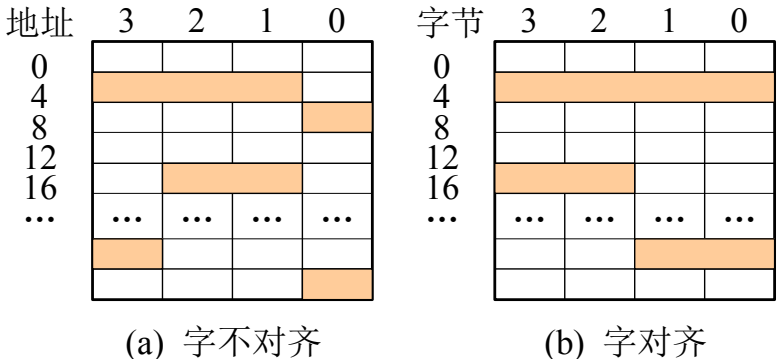


图 2-8　存储器中字的对齐情况

字不对齐的存储方式可以避免数据存储中的空白，可以节省存储空间。所以某些计算机允许字不对齐存储。在一些较新的计算机中，则既支持对齐的数据字存储方式，又支持不对齐的数据字存储方式。

2.5.2. 数据的寻址方式

操作数据的寻址方式是指令的地址码字段中存放的内容及其编码方式。程序员看到的操作数在存储器中的地址是逻辑地址；因此，操作数的寻址方式就是地址码形成操作数的逻辑地址的方法。在不同寻址方式下，地址码表示不同的含义。在某些计算机中，一条指令中的某一个地址码只有一种固定的寻址方式，不同的寻址方式出现在不同的地址码中。而在有的指令系统中，一条指令中的每一个地址码都可以有多种不同的寻址方式。指令中需要表示每一个地址码的寻址方式。这时指令中的地址码由寻址方式特征位和形式地址两部分组成。

在上节的例子中，已经讲到了指定内存地址和指定寄存器号这两种指定操作数存储位置的方法。在指令中指定操作数存储位置的方法还有很多种。在指令中提供不同寻址方式的目的在于使得程序中能更加灵活地指定操作数的存储位置。特别是在重复执行的程序段中以及对数据结构进行操作的程序中，使得这些程序能够不加修改地完成对各种结构的数据元素的寻址。

在计算机中常用的寻址方式有以下几种。

1. 隐含寻址方式

在指令中不指出操作数的地址，根据指令的操作码就可判定操作数的存储位置，即操作数的地址隐含在操作码中。如对于存放在堆栈中的数据或者对累加器中的数据进行运算操作的指令。

2. 立即数寻址方式

操作数直接在指令中给出，如图 2-9 所示。这种寻址方式可使操作数与操作码一起读取，指令读取的时候就把数据从存储器中读取了，数据作为指令的一部分放在了指令寄存器中，从而可节省一次读取操作数的访存操作，因而加快了指令执行速度。但是由于一般规定程序在运行中不允许改变，因此立即数寻址方式指定的操作数在程序执行过程中是不能改变的。这种寻址方式适合于访问一些常数。在汇编语言中为表示立即数寻址，通常直接用数字表示，在有些计算机中，在数字前还加上一个#号或其它符号，以便与地址值相区别。

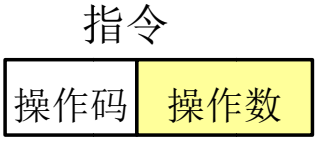


图 2-9 立即数寻址

3. 寄存器寻址方式

操作数在通用寄存器中，寄存器号在指令中给出，如图 2-10 所示。一般计算机中都有存放操作数的通用数据寄存器，指令中只需指定寄存器号，所需的操作数可从这个寄存器中得到。因为通用寄存器的数量一般很少，因此只需少量的代码指定，从而减少了整个指令的长度。同样由于通用寄存器的数量少，所以不可能将所有的数据都安排在寄存器中。在汇编语言中为表示寄存器寻址，一般用寄存器名，如 R1、R2，或者＄1、＄2 等。

4. 直接寻址方式

操作数在存储器中，它的地址直接在指令中给出。指令中的地址码直接用于访问存储器，如图 2-11 所示。这种寻址方式简单、直观，也便于硬件实现，但地址码的长度依赖于存储空间的容量。随着存储空间的不断扩大，所需的地址码也越来越长，使指令的长度也相应增加。此外，操作数的地址是指令的一部分，不能修改，因此这种寻址方式适合于在一条指令中访问固定存储器单元。在汇编语言中，为表示直接寻址方式，一般用直接用数字表示地址值，如 1000、2000 等，或者在地址值前面加上一个符号。

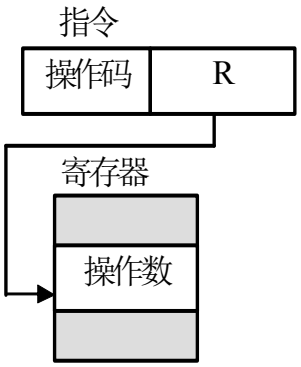


图 2-10　寄存器寻址

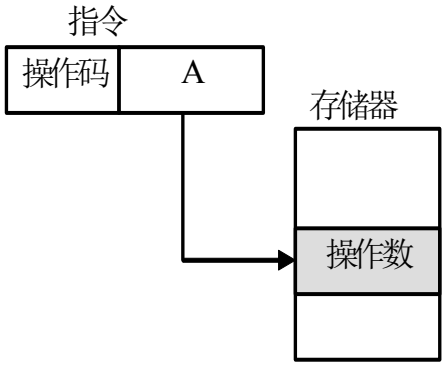
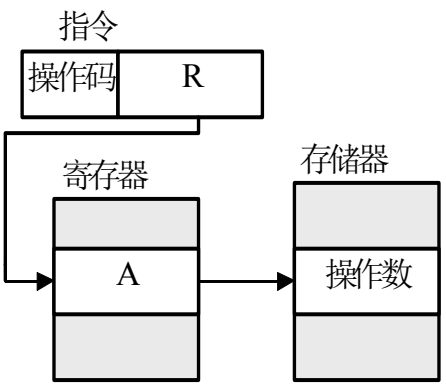


图 2-11 直接寻址

5. 间接寻址方式

操作数的地址在寄存器中或者在主存储器中，又分别称为寄存器间接（见图 2-12）和存储器间接寻址（见图 2-13），其寄存器号或存储器地址在指令中给出。在寄存器间接寻址方式下，指令中指定一个寄存器号，寄存器中的内容是操作数在存储器中的地址。寄存器间接寻址方式可方便地对数组中的数据元素进行寻址，指令中可以用较少的代码位来指定操作数的地址。在存储器间接寻址方式中，指令中给出的既不是操作数，也不是操作数的地址，而是操作数地址的地址，这称为一级间址。此外还可以有多级间址。间接寻址方式有很强的寻址能力，可方便地对复合数据类型中的数据元素进行寻址，但是执行时需要多次访问存储器才找到操作数，降低了指令的执行速度。在汇编语言中，为了表示寄存器间接寻址，一般在寄存器名外加上括号。如（R1）、（R2）等，表示根据 R1 或 R2 中的地址从存储器中访问数据。在汇编语言中为表示存储器间接寻址，一般在地址值外加上括号，如（1000）、（2000）等，表示从地址为 1000 或 2000 的存储器单元中读取数据的地址，然后根据这个地址访问数据。



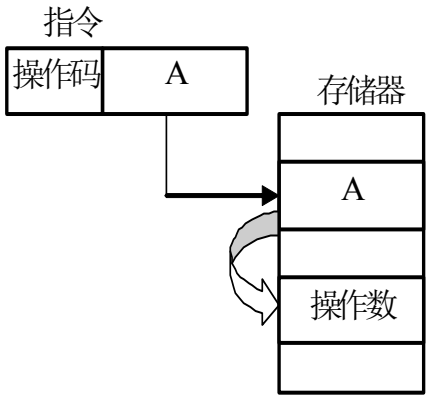
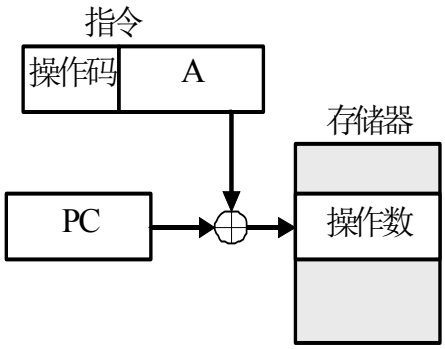


图 2-12　寄存器间接寻址　　图 2-13 存储器间接寻址

为了支持对数据的顺序访问，还可以采用带有自动增量和自动减量的寄存器间接寻址方式。即在访问之后自动修改寄存器中的值，使其指向下一个数据元素的地址。

6. 相对寻址方式

操作数的地址是程序计数器 PC 的值加上一个偏移量，这个偏移量在指令地址码中给出，如图 2-14 所示。因为访问的数据在存储器中的存储位置相对于指令的位置是固定的，因此称为相对寻址方式。这种寻址方式下，被访问的操作数的地址是不固定的，在程序中，不同的地方用相同偏移量的相对寻址方式访问的数据存储位置是不同的。不管程序装入到内存中的什么位置，访问的数据的位置相对于程序的位置都一样。指令中给出的偏移量可以是正值，也可以是负值，通常用补码表示。偏移量的长度一般要比存储器地址的长度要短，如 16 位，所以相对寻址方式可以用较短的地址码访问存储器。在汇编语言中为表示相对寻址方式，一般在字符 PC 外加上括号和偏移量的值，如 100（pc）、－200（pc）等。



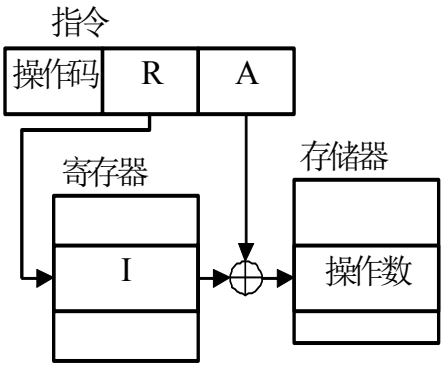


图 2-14　相对寻址　　图 2-15　变址寻址

7. 变址和基址寻址方式

指令地址码中给出一个存储器地址值，操作数的地址由指令地址码与某个寄存器的数值相加形成，如图 2-15 所示。在变址寻址方式中，这个寄存器称为变址寄存器，它可以是一个专用的变址寄存器，也可以是通用寄存器。变址寄存器也必须在指令中指定。这种寻址方式适合于对一组数据进行访问，这时指令给出的地址位是数组的起始地址，寄存器中的值是数组的下标。在访问了一个数据元素之后，只要改变变址寄存器的值，就可形成另一个数据元素的地址，指令本身不变。这样便于用一个循环程序来对数组进行处理。在汇编语言中，为表示变址寻址方式，一般在寄存器名外加上括号以及偏移量的值，如 100（R1）、－200（R1）等。

在基址寻址方式中，操作数的地址是一个称为基址寄存器的值加上指令中给出的地址偏移量。当存储器容量很大，而导致访问存储器的地址码太长时，通常的解决办法是将整个存储空间分成若干个段，段的起始地址存放于基址寄存器中，段内的偏移量由指令中的地址码给出。基址寄存器中包含完整的地址，地址位数较多，地址偏移量的位数可以是较少的。将偏移量与基址寄存器的值相加就构成访问存储器的地址。基址寄存器中的内容在进程中一般不变，要访问不同的数据可以改变偏移量的值。基址寻址方式用于虚拟存储器的地址映像、程序定位、扩大寻址范围等。基址寄存器号可以在指令中指定，也可以是隐含的。

8. 复合寻址方式

复合寻址方式是把多种寻址等方式相结合而形成的寻址方式。如把间接寻址方式同相对寻址方式或变址寻址等方式相结合而形成的寻址方式。它分为先间接方式与后间接方式两种。通过寻址方式的复合可以增强指令的寻址能力，或者在相同寻址能力下减少基本寻址方式的数量。

总之，不同的寻址方式有不同的特点和用途。增加指令的寻址方式方便了汇编语言的程序设计，可以提高程序代码的效率。但是较多的寻址方式使得指令的格式较为复杂，指令的执行控制也变得较为复杂。

2.6. 指令系统

以上介绍了指令的编码，本节将介绍指令系统的设计。先介绍指令系统设计中需要考虑的基本原则，已有的计算机中的指令系统，指令系统设计的思想和理念。

2.6.1. 指令系统的设计

指令系统是软件与硬件的界面。指令系统的设计是指对于指令类型的选择和指令格式的确定，即确定在计算机中设置那些指令以及指令的编码。它是计算机系统设计的关键。指令系统的设计需要考虑到计算机功能的完整性、指令的有效性、可扩展性和兼容性。

功能的完整性就是计算机的指令系统能够覆盖计算机所需要的各种功能，任何设计功能都可以用指令系统中的指令编写的程序有效地完成。一些基本的功能应当在指令中直接提供，特别是常用的功能，如加、减、乘、除运算，访问存储器的操作等。某些不常用的功能可以用基本的指令功能组合实现，也就是用程序实现。

指令的可扩展性就是要保留一定余量的操作码空间为以后的功能扩展所用。一般来讲，在系列产品中作为后继产品的计算机系统，总要在先前的系统基础上增加一些指令。

指令系统的有效性是指利用该指令系统所编写的程序能够高效地运行。主要表现在程序代码占用的存储空间小，执行速度快。实现指令系统的有效性不仅要求指令的编码高效紧凑，更重要的是使得指令系统的功能与实际应用的操作相匹配，以便用最少的指令完成所需的功能。

指令的兼容性是指机器指令的通用性。指令的兼容使得已有的软件能够得到利用。为了实现指令的兼容，计算机系统可设计成系列化的。系列化的计算机是指基本指令系统相同、基本系统结构相同的一系列计算机系统产品。系列机各机型之间具有相同的基本指令集，因此指令系统是兼容的。

2.6.2. 常见指令类型

每一种计算机都有各自不同的指令集合。在这些指令集合中，有些类型的指令是各种计算机共同的，有些是各不相同的。常见指令类型包括以下几种。

1. 数据传送指令

将数据在主存与 CPU 寄存器之间进行传输，包括取数（load）指令、存数（store）指令、存储器之间数据传送（move）指令、寄存器之间数据传送指令、成组数据传送指令等。load 指令将存储器中的数据读出后放入寄存器中，store 指令则将寄存器中的数据写入存储

器中。有些计算机中采用 move 指令，它可以将数据在存储器和寄存器之间任意传送，包含了 load 和 store 指令的功能。

2. 算术运算指令

对数据进行算术操作，包括加法、减法、乘法、除法、算术比较等。算术运算指令还可以分为定点数的运算指令和浮点数的运算指令，定点数的运算指令可以针对不同数据位数有不同的指令，浮点运算指令也可以分为单精度运算和双精度运算两类。因为计算机的定点数运算总有精度限制，对较长的数据可以进行分段的运算。例如 16 位字长的计算机中要进行 32 位定点数据的加法运算时就要分两次进行，先进行低 16 位的加法运算，再进行高 16 位的加法运算，中间用进位标志传递进位信息。

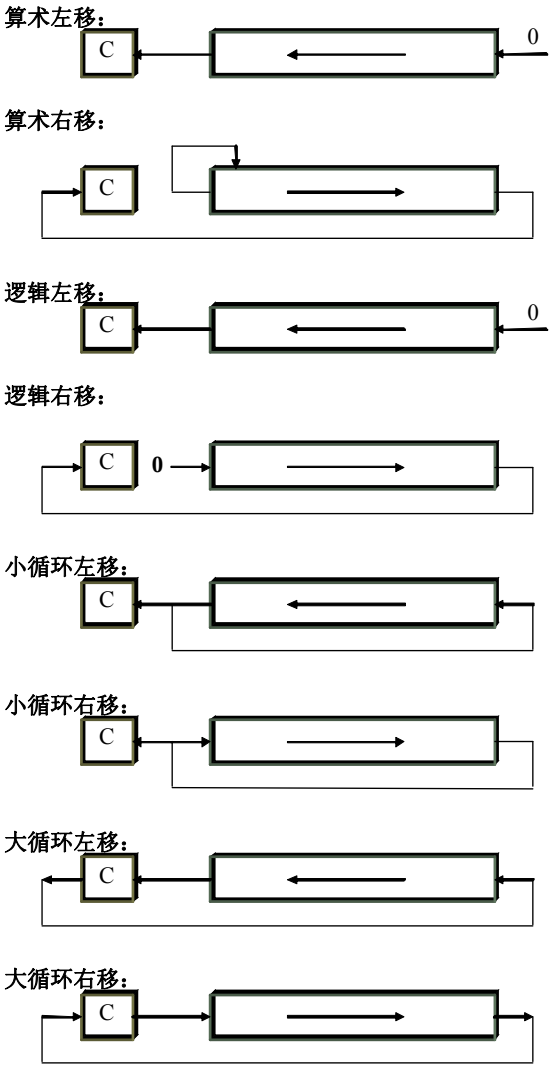


图 2-16　移位操作类型

3. 逻辑运算指令

对数据进行逻辑操作，包括逻辑加、逻辑乘、按位求反、求补、移位和数据格式转换等。逻辑加是一种按位或操作，逻辑乘是一种按位与操作。这些逻辑运算分别处理数据中的各个位，位与位之间没有进位传递关系。

移位操作在相邻的数据位之间进行传递操作，分为算术移位、逻辑移位和循环移位三种，每种移位操作又有左移和右移之分。算术移位可用于实现乘除法的运算。算术右移保持最高位（符号位）不变，相当于符号扩展，而逻辑右移最高位补 0。循环移位可以将进位标志一起参与到移位操作中，构成大循环；也可不包括进位标志位，构成小循环。这里的进位标志位就是在状态寄存器中的一个状态位，通常用于记录加法操作的最高位进位输出。在大循环移位中，从寄存器移出的位放入进位标志位 C，原来的进位标志位 C 移入到寄存器中。在小循环移位中，寄存器移出的位放入进位标志位 C，同时从另一端移入寄存器，原来的进位标志让 C 丢弃。图 2-16 是各种移位操作情况的示意图。

4. 程序流控制指令

改变指令执行的顺序，包括条件转移指令、无条件转移指令、转子程序指令、子程序返回指令、软中断指令、中断返回指令和空操作指令等。

条件转移指令一般与比较指令配合使用。在较早的计算机中，将比较和转移功能分成两条指令，比较指令结束后对相应条件码进行置位，然后由后面的条件转移指令判断是否满足

转移条件。一些新型计算机中将这两条指令合并成一条指令，使其同时具有比较和转移这两个功能。将这两条指令合并成一条指令后，该指令将直接对两个操作数进行比较，然后根据比较结果判断是否进行转移。这样就可省去一条指令，而且不需要保存比较的结果，可省去条件状态码寄存器。

无条件转移指令不作条件判断，只是修改 PC 的值，以将程序转移到新的位置。计算机中通常把无条件转移指令用助记符 JUMP 表示，把条件转移指令用 BR（branch）表示。转子程序指令除了具有程序转移的功能外，还将当前 PC 的值放入堆栈，以便于程序的返回。子程序返回指令从堆栈中弹出 PC 值，实现从子程序中的返回。软中断指令主要用于调用操作系统提供的子程序，即利用操作系统提供的异常处理机制实现对操作系统子程序的调用。它在调用操作系统的子程序时将系统的状态改变为系统态，从而具有更高的优先级别，实现应用程序不能实现的功能，也简化了应用程序的设计。中断返回指令使得程序返回时系统状态恢复为用户状态，从而继续执行中断前的程序。空操作指令不作任何操作，只是控制程序的执行时间，可用于指令流水中的时间控制，以避免因指令流水执行而可能带来的时序上的错误。

5. 输入 / 输出指令

用于启动外围设备、检查测试外围设备的工作状态、读写外围设备的数据。有些计算机采用专门的输入 / 输出操作指令，而有的计算机则没有专门的外设操作指令，它们把外设控制器看成是一个特殊的存储器单元，因而用访问存储器的指令访问外围设备的接口。

6. 堆栈操作指令

堆栈是一种按特定顺序进行访问的存储区。这个存储区中数据的个数和内容随着对其访问而动态地变化，访问只在区域的一端进行。堆栈的主要操作是压栈（push）和出栈（pop），压栈操作将数据写入堆栈的一个新的单元，出栈操作将一个数据从堆栈中取出，并释放占用的存储单元。先进入堆栈的数据总是后弹出堆栈，这种特定的数据访问特性称为后进先出（LIFO）。堆栈操作广泛用于程序在调用于程序时的数据保存和恢复。在某些计算机中，堆栈还用于存放运算的操作数。这些计算机中需要有堆栈操作指令。

7. 字符串操作指令

这是一种非数值运算的指令，用于文字处理的应用场合，一般包括字符串传送、字符串转换、字符串比较、字符串查找、字符串匹配、字符串抽取和替换等。这些操作广泛应用于文字处理、数据库查询等商业应用等场合。字符串处理指令可以加快这些应用软件的速度，所以一些计算机中设置了字符串处理指令。

8. 多媒体指令

图形数据等多媒体数据的运算由多媒体指令实现。多媒体指令已成为桌上型计算机和嵌入式计算机系统普遍采用的指令类型。许多音频和视频数据的处理都可以利用多媒体指令实现，从而加快音频和视频数据的处理速度。随着数字信号处理（DSP）技术的推广应用，一些计算机中还出现了支持 DSP 的指令，这些指令能够进行高速乘加运算，从而加速典型 DSP 算法的计算速度。

9 系统指令

系统指令用于改变计算机系统的工作状态，改变系统配置等。系统指令类型如改变执行的特权、进入特殊的处理程序、Cache 初始化、设置大小数端、CPU 功能配置等。系统指令

常用于在操作系统中对系统资源进行配置、初始化和访问控制，这些系统资源对应用程序员可能是透明的，如虚拟存储器中页面失效的处理等。系统指令只有在操件系统状态下才能执行。在不同的计算中，系统指令的设置差异较大。许多操作系统都要求计算机提供一些系统指令，以支持操作系统的特殊操作类型，并对这些操作实施保护措施。

以上几种指令中，前四种指令是必备的指令，后五种指令并不是必备的指令。例如，有些计算机中没有字符串处理指令，有些计算机中没有堆栈操作指令，一些较为简单的计算机中没有系统指令。

程序是由多条指令构成的，这些指令的存储可以按执行顺序安排，也可以不按执行顺序安排。按执行顺序安排是最简单直观的方式，因此程序中大多数的指令一般都是按执行顺序安排的。在按执行顺序安排指令存储顺序的方式中，我们用一个程序计数器 PC 表示执行指令的存储位置。因为下一条指令紧挨在上一条指令的后面，它的地址是上一条指令地址码加上指令的长度值（字节数）即可生成。但是程序中的指令执行不会全都按存储顺序进行，程序流程中存在大量循环结构和分支结构。为了形成分支的程序流程，需要采用转移指令来形成非顺序的下一条指令的地址。采用转移指令可以实现程序分支或构成循环程序，或者子程序的调用，从而能缩短程序长度，提高程序代码效率。

转移指令的功能就是形成下一条指令的地址，并将下一条指令的地址写入 PC。它将 PC 更新为转移目标指令所在的地址值。转移指令可分为无条件转移指令和条件转移指令两种。条件转移指令是根据计算机中的状态决定是否转移，这些状态包括是否有进位、操作的结果是否为零、操作的结果是否是负数、数据是否溢出等。这种状态可以存放在某个通用寄存器中，也可以存放在专用的寄存器中。在一些 CPU 中专门设置一个专门存储这些状态的寄存器，称为状态寄存器 SR。状态寄存器中用不同的字段记录不同的运算状态，各种状态条件在指令中通常也用助记符表示，如：

N（negative）：如果结果为负数则设置为 1，否则清 0。

Z（zero）：如果结果为零则设置为 1，否则清 0。

V（overflow）：如果结果数据溢出则设置为 1，否则清 0。

C（carry）：如果结果产生了进位则设置为 1，否则清 0。

P（Parity）：如果结果中有奇数个 1 时为 1，有偶数个 1 时为 0。

这此状态位的组合可产生新的状态表示，如大于或等于（不小于）、小于或等于（不大于）。条件转移指令前一般用一个比较操作形成状态条件，供条件转移指令使用。这两条指令必须连续执行，不能被隔开或者改变执行顺序。在近年出现的具有比较功能的条件转移指令中，指令产生转移条件之后立即根据条件进行转移，而不是存储在寄存器中。

根据形成转移目标指令地址的方法，转移指令还可以分为绝对转移指令和相对转移指令两种。在绝对转移指令中，转移目标的指令地址作为地址码在转移指令中给出。相对转移指令则给出一个地址偏移量，即相对于当前指令的地址偏移量，在形成目标地址时需要将当前 PC 的值加上指令中给出的地址偏移量，从而形成转移目标的地址。

在一些嵌入式处理器的应用中，经常需要进行数字信号处理。为此，一些嵌入式处理器设置了 DSP 指令。DSP 运算中的一个常用操作是乘法－累加的运算。其中的乘法一般是较短的数据，如 16 位定点数，而累加运算则一般是较长的数据，如 32 位或 64 位定点数。这种乘法－累加运算用于实现数字滤波器。在通用计算机中，实现这个运算由一个循环程序构

成。为了有效地实现这种乘加运算，处理器必须能够有效地进行乘法运算。一些支持 DSP 的处理器中通常由一条 MAC 指令来实现。专用的 DSP 处理器则可以用一条指令来实现上述循环。

2.6.3. 指令系统设计思想

在计算机的发展过程中，由于不同的设计理念，形成了各种不同风格的指令系统。了解这些指令系统的特点有助于设计和选择新的计算机指令系统。计算机指令系统的类型可以分成复杂指令系统计算机（CISC）和精简指令系统计算机（RISC）两种类型。早期 CISC 设计风格的主要特点是：

（1）指令系统复杂。具体表现在指令数多、寻址方式多、指令格式多。CISC 的指令数一般大于 100 条，寻址方式一般大于 4 种，而且每个地址码都有多种寻址方式，指令格式一般大于 4 种。

（2）指令串行执行。将每一条指令的执行分成许多执行步骤，绝大多数指令需要多个时钟周期才能执行完成。

（3）各种指令都可访问存储器。由于每个地址码都有多种寻址方式，各种指令都可访问存储器，使得同一条指令有许多不同的执行步骤，需要进行不同的控制，指令的执行步骤和执行时间变化范围很大。

（4）有较多专用寄存器。如变址寄存器、各种段基址寄存器、状态寄存器等，寄存器的编码和使用较为复杂。

（5）编译程序难以用优化措施生成高效的目标代码程序。因为指令系统复杂，编译程序面临复杂的选择，在代码优化阶段难以预料生成的指令代码的情况，因而难以做出有效的优化选择。

在早期的计算机设计中，认为指令数量越多就性能越好。因为指令越多意味着硬件的功能越多。对于某一项功能，用硬件实现总是比用软件实现的速度快。所以早期计算机中把指令的数量作为计算机性能的一项重要标志。

随着指令功能的增强和指令的数量不断增加，计算机指令系统越来越复杂，从而使得计算机的控制器也越来越复杂。许多早期计算机的指令数已经达到 200 条以上，有些指令的功能异常复杂，需要使用多种不同的寻址方式、指令格式和指令长度，指令的执行流程复杂，执行时间很长，所以这种计算机被称为 CISC。在微处理器发展的初期，CISC 面临的主要问题是控制器电路过于复杂，占用了微处理器芯片中过多的面积。更重要的是，CISC 的复杂控制器影响了 CPU 时钟频率的提高，指令的复杂控制方式不便于采用流水技术等提高性能的措施，使得性能提高较为困难。此外，复杂的寻址方式增加了访存的次数。随着 CPU 工作频率的提高，访存操作相对显得越来越慢。

针对 CISC 中存在的问题，美国加州大学伯克利分校的 Patterson 教授于 1979 年提出了 RISC 的概念，通过简化指令系统和简化控制器来寻找提高系统性能的方法，并先后研制成 RISC－1 和 RISC－2 计算机。1981 年美国斯坦福大学在 Hennessy 教授的研究小组研制成 MIPS 计算机。在此基础上又开发出了 SPARC 和 MIPS 系列的 RISC 微处理器。这些 RISC 处理器通过简化指令系统，简化了计算机中指令译码器和控制电路，使得运算速度得到了提高。

早期 RISC 技术的主要特征归纳起来有以下几点：

（1）简化的指令系统。表现为指令数较少、基本寻址方式少、指令格式少、指令字长度一致。在 RISC 中，选择使用概率很高而简单的指令，以及虽然功能较复杂但使用概率较高且又能较好支持高级语言和操作系统实现的指令来组成指令系统。初始系统的指令总数大都不超过 100 条，寻址方式一般限制在两三种，而且每个地址码的寻址方式是固定的，指令格式一般限制在两三种，指令字长度一般为 32 位。简单的指令格式使得指令的译码简单，也使得指令执行的控制变得简单。

（2）以寄存器－寄存器方式工作。即指令系统中除 LOAD／STORE 指令可访问存储器外，其余指令都只访问寄存器。采用这种工作方式的目的是减少对存储器的访问。随着 CPU 和存储器工作速度差异的不断扩大，减少访存对于提高计算机的性能显得越来越重要。

（3）采用流水技术。在 RISC 中，除 LOAD／STORE 指令外，其他指令都以流水方式工作，下一条指令不必等到上一条指令执行完毕才开始执行，而是可以提前执行，指令的执行相互重叠，从而平均可在一个时钟周期执行完毕一条指令。指令的流水执行方式显著提高了计算机的性能，RISC 简单的指令功能和固定的执行流程使得流水执行方式成为可能。在后来的 RISC 技术中，不仅将 LOAD／STORE 用流水方式执行，而且形成了各种先进的指令流水执行技术。

（4）使用较多的通用寄存器以减少访存。一般至少有 32 个通用寄存器，不设置或少设置专用寄存器，以支持寄存器－寄存器的工作方式。在采用先进流水技术的 RISC 中，设置了大量硬件内部寄存器，存储数据中间结果，并且由硬件对寄存器的使用进行分配，从而避免寄存器使用的冲突，提高指令流水线的工作效率。

（5）采用优化编译技术。通过精心选择的指令系统，并采用软件手段，特别是优化编译技术，力求能有效地支持高级语言实现，能容易地生成优化的目标代码，防止或减少指令流水线中出现的相关性，以保证指令执行流水线畅通。RISC 机中所采用的编译技术突出了两点：一是对寄存器分配进行优化，以减少对存储器的访问；一是设法对程序中的指令序列在保持原来语义的基础上进行重新排序和调度，以提高程序的执行速度。

判断一个计算机是否属于 RISC 型的标准，并不是上述五个条件都必须满足。有的计算机虽然不具备上述某些特征，但从总体上来讲仍属于 RISC 计算机范畴。RISC 与 CISC 技术两者的主要区别在于设计思想和理念上的差别，而在指令设计风格上反映出来。RISC 计算机在系列化发展中也会不断增加新的指令，使得指令的数量越来越多。但是 RISC 设计思想在增加新指令的同时更加全面地考虑性能的提高，综合评价和权衡新的指令对系统总体性能的影响。

采用 RISC 设计思想的计算机产品有 Digital 公司（现属惠普公司）的 Alpha 系列处理器、惠普公司的 PA－RISC 系列、IBM 和摩托罗拉公司的 PowerPC 系列、SGI 公司的 MIPS 系列和 SUN 公司的 SPARC 系列。此外，还有一些嵌入式处理器，如 ARM 公司的处理器系列等。

第3章. 存储系统

计算机的工作依赖于存储器中的程序和数据,构成主存储器的基本元件是存储器芯片,计算机的主存储器主要由随机访问的存储器芯片构成,存储器芯片与其他部件之间的连接通过数据线、地址线和控制线进行。本章将介绍存储器的基本工作原理、存储器设计方面的基本概念和方法，以及用存储器芯片构成存储器的方法等，首先从存储器芯片的结构讲起。

3.1. 存储器芯片简介

半导体的存储器主要有随机存储器（RAM，Random Access Memory）和只读存储器（ROM，Read Only Memory）两大类，半导体随机访问存储器(RAM)芯片主要有静态存储器（SRAM，Static RAM）芯片和动态存储器（DRAM，Dynamic RAM ）芯片两种。静态存储器芯片的速度较高,但它的单位价格即每字节的价格较高;动态存储器芯片的容量较高，但速度比静态存储器慢。

3.1.1. SRAM 芯片的结构和工作原理

静态存储单元的结构有多种，一种典型的 CMOS 静态存储单元如图 3-1 所示。它是一个触发器结构，存储的数据表示为由晶体三极管 T1 和 T2 构成的双稳态电路的电平，电路中 T3 和 T4 是与 T5 和 T6 互补的晶体管，T5 和 T6 用于将存储单元与外部电路连接或者隔离，由行选通信号控制。在图 3-1 中，VCC 为电源正极，VSS 为信号地。当 T1 导通时，A 点将出现低电平，这个低电平送到的栅极，使得 T2 截止；T2 的截止使得 B 点出现高电平，这个高电平连接到 T1 的栅极，从而使得 T1 导通，这样，T1 导通 T2 截止的状态就是一个能够持续保持的稳定状态。同样，T2 的导通 T1 截止的状态也是一个稳定状态。这是一个双稳态电路，两种稳定状态分别可用于存储数据 0 和 1，就像一个触发器一样。在这个存储单元中，如果要将存储的信息读出来，就要在行选通信号线上施加高电平，使得晶体管 T5 和 T6 导通。这样存储的信息就可以通过 T5 和 T6 输出到数 据线 D 和 $\overline{D}$ 上。这两条数据线是一对互补的数据信号线，连接一列存储单元的数据端，并将某一个存储单元的数据输出到存储器外部。如果要将外部的数据写入存储单元，则外部电路驱动数据线，使得外部的信息输入到存储单元，改变存储单元的导通和截止状态，从而将信息写入存储器。数据线既是数据写入线，又是数据读出线，在每条数据线上有一个数据读写电路。在这种静态存储器的存储单元中，需要 6 个 MOS 管来存储一位信息。

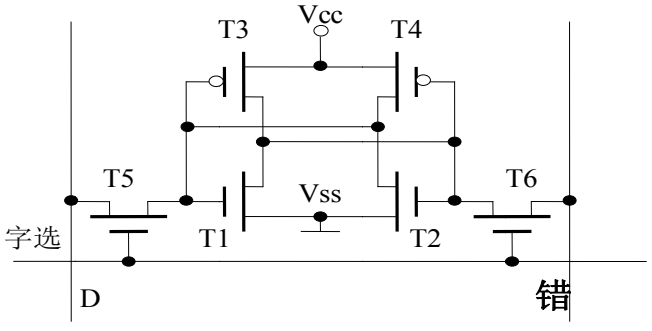


图 3-1 CMOS 静态存储单元结构

将大量这样的存储单元合起来可以构成一个存储单元阵列，存储大量的信息。在存储器芯片中包括存储体、读写电路、地址译码电路和控制电路等组成部分，如图 3-2 所示。存储体部分由大量的存储单元构成的阵列组成。在阵列中包含许多行，每行包含许多个存储单元，构成许多列。为了对各个不同的存储单元进行访问，阵列中用一条行选通线和一条列选通线选择阵列中的单元。行选通线用于选择一行中的存储单元，将一行中的各存储单元的数据读出。为了对芯片中的某一个存储单元进行访问，还需要形成列选通线，以选择一行中的某一个存储单元。列选通线既是数据写入线，又是数据读出线，因此在每条列选通线上有一个数据读写电路。

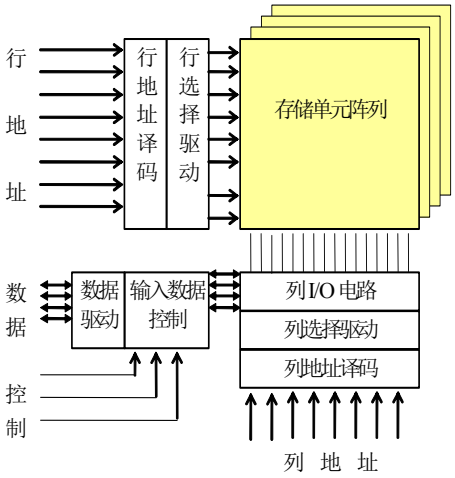


图3-2 RAM的阵列结构

地址译码器的输人信号线是访问存储器的地址编码，它将二进制代码形式的地址转换成驱动读写操作的选择信号，以便选择所要访问的存储单元。地址译码的方式有两种可能，一种是单译码方式，另一种是双译码方式。在单译码方式的存储器中，只用一个译码电路，将所有的地址信号转换成行选通信号。在一行内的各存储单元构成一个数据字的存储单元。每一列对应于一个数据位。在一个具有 10 位地址的存储器芯片中，单译码方式的译码器产生 1024 条行选通信号线，每一条行选通信号线选择一个字对应的存储单元。这种方式适合于容量小的存储器芯片。在容量较大的存储器芯片中，一般采用的是双译码的方式，将存储单元构成一个方阵，外部输入的地址线分为行地址和列地址两部分。通常访存地址的高位部分作为行地址，低位部分作为列地址。地址译码器分为行地址译码和列地址译码两个部分，分别产生行选通信号和列选通信号，行选通信号和列选通信号都有效的存储单元被选中，即如图 3-2 所示的方式。行选通线和列选通线 中都只有一条线是有效的，阵列中一行和一列的交叉点上的单元被选中。其中的数据被读出，或者将外部的数据写人其中。这样每个译码器都比较简单，可减少数据单元选通线的数量。例如，对于一个 10 位地址的存储器芯片，双译码方式用两个译码器各从 5 位地址信号中译码生成 32 条选通线，因此一共产生 64 条选通线，大大少于单译码方式的情况。在这种译码方式中，存储器芯片中的一个阵列构成一个 1 位数据的存储结构，在多位的存储器芯片中，就要有多个这样的阵列。每个阵列构成一位数据的存储空间，采用相同的行选通和列选通信号。访问时将数据从每个阵列的同一行同一列读出或者写入。

由于选通信号线要驱动存储阵列中的大量单元，因此需要在译码器之后加一个驱动器，用驱动输出的信号去驱动连接在各条选通线上的各存储单元。数据驱动电路对读写的数据进行读写放大，增加信号的强度。

输入/输出电路处于存储器芯片的数据线和被选用的单元之间，用以控制被选中的单元读出或写入，并具有放大数据信号的作用。数据驱动电路将数据驱动后输出到芯片外部，有些芯片中采用分离的数据输入信号和输出信号，有些芯片则将数据输入与数据输出信号合并，形成双向的数据线。

静态存储器存储数据的原理与寄存器是一样的。用静态存储器芯片构成的存储器就像是一个容量很大的寄存器组，对它的访问控制十分简单。一般的地址线和数据 线可直接连接到静态存储器上，不需要专门的控制电路控制静态存储器的访问操作。

SRAM 芯片的引脚接口信号通常有：

ADD：地址信号，在芯片手册中通常表示为 A0，A1，A2，…。

CS*：芯片选择，低电平时表示该芯片被选中。

WE*：写允许，低电平表示写操作，高电平表示读操作。

DOUT：数据输出信号，在芯片手册中通常表示为 D0,D1,D2，…。

DIN：数据输入信号，也表示为 D0,D1,D2，…。

OE*：数据输出允许信号。

存储芯片中的控制电路用于控制芯片的操作。它根据外部提供的控制信号进行控制，外部提供的控制信号如读写控制、片选控制、输出控制等，这些控制信号一般分别称为 WE*, CS*和 OE*。这里信号名中的星号(*)表示该信号是低电平有效的，有时还用信号名上的一横（求非符号）或者#号表示信号的低电平有效。读写控制信号 WE*指定操作的方式。不同的存储器芯片产品有不同的控制信号名称，以及信号的有效电平表 示，如 CS*信号也常表示为 CE* (芯片许可，Chip Enable)，或者表示为高电平有效的 信号（CE 或 CS）。这样构成的存储器芯片对于任何地址的访问都具有相同的访问时间，在没有写操作的情况下，数据能够始终保持不变，因此称为静态随机访问存储器（SRAM）。

以上介绍的是存储器芯片的物理结构。在逻辑上，通常将存储器芯片的容量特征表示为字数与位数的乘积，字数代表存储器芯储阵列的规模，位数表示数据宽度，例如，一个 32 行 32 列存储阵列的存储器芯 数为 32×32 = 1024，如果数据宽度是 4 位，则这样的存储器芯片逻辑上表示为 1024×4 (或 1K×4)。存储器芯片的字数影响到芯片所需的地址数据宽度则对应着芯片的数据线数量。一个 1K 位的存储器芯片有 10 条地址线（210=1024）数据线。静态存储器芯片的电路图表示如 3-3 图示，该芯片是一片 8K×8 位的 SRAM，图中标出来每个信号线引脚的名称和位置编号，其中 $\overline{CS_1}$ 为低电平有效的芯片选择信号，CS2 是高电平有效的片选择信号，两个芯片选择信号之间是“与”的关系，必须都有效时，芯片才工作。此外，SRAM 芯片还有写许可和输出许可控制信号。

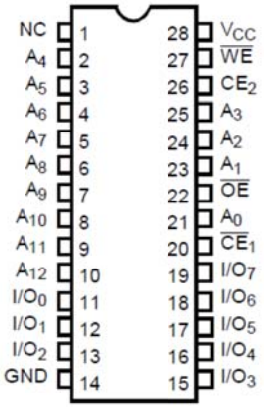


图 3-3 静态 RAM 芯片引脚电路

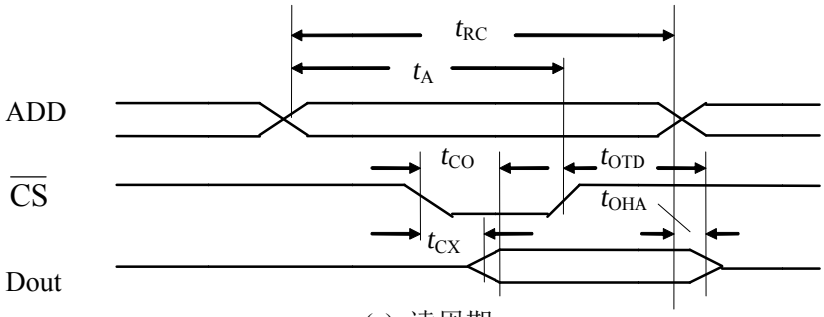
SRAM 芯片的操作时序如图 3-4 所示，其中的主要参数有：

t_A：访问时间，即从地址信号输入到数据读出的延迟时间。

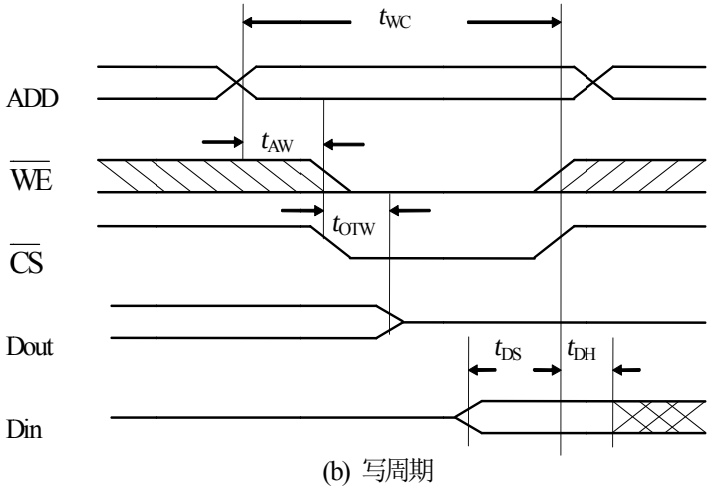
t_{RC}：读周期时间，即从一个读操作开始到下一个访问操作开始的间隔时间。

t_{WC}：写周期时间，即从一个写操作开始到下一个访问操作开始的间隔时间。

t_{OHA}：地址改变后的数据保持时间。



(a) 读周期



(b) 写周期

图 3-12 静态存储器的读写周期

在时序图中，用信号波形图来表示操作时序，其中上下两条水平线表示地址线和数据线的状态，因为它是一束信号线的情况。在这一束信号线中，有的信号可能处在高电平，有的信号线可能处在低电平。存储器的数据线和地址线实际上是一种总线。在总线波形图中，稳定的总线信号用上下两条水平线表示，用交叉线表示总线信号内容的改变。数据总线以及地址总线上的信号总是一起发生变化的。因为总线是由多条信号线构成的，我们一般并不关心总线中每条信号线是高电平还是低电平，而只是关心总线中的各条信号线什么时候保持不变，什么时候发生变化。时序图中阴影线部分表示信号可以任意变化。总线信号还有一种状态是截止状态，也就是各信号输出处于高阻抗状态，这种状态一般用一条处于中间位置的水平线表示。各种 ROM 存储器芯片的控制时序与静态 RAM 芯片类似，但是 ROM 芯片的引脚上没有读写控制信号，因为它只有读操作。

从 SRAM 的操作时序图上可以看出，SRAM 存储器芯片要求外部电路在读取 SRAM 数据的过程中，首先驱动地址线，然后将 CS*信号和 OE*信号置低电平，使得存储器芯片进行操作。存储器芯片将数据从存储阵列中读出后驱动数据输出线，这时外部电路就可以采集数据了。在这个过程中，应保持高电平。

在向 SRAM 写入数据的过程中，要求外部电路首先也是驱动地址线，然后将 WE*控制信号和 CS*信号置低电平，OE*信号置高电平，使得存储器芯片进行写操作。然后驱动数据线，将需要写入的数据送往存储器芯片。经过一定的延迟之后，数据线上的数据信号就写入到地址线信号所指定的存储位置中。

存储器芯片读周期时间一般与写周期时间相同，统称为访问周期时间。在计算机的工作过程中，周期性地访问存储器。存储器读操作的周期时间也称为读周期，写操作的周期时间也称为写周期。存储器的读周期和写周期又统称为访问周期。

3.1.2. DRAM 芯片的结构和工作原理

利用 MOS 晶体管的高阻抗特性和电容器可以直接构成存储单元。这种存储单元利用电容器存储电荷的特征来存储数据，用 MOS 管控制数据的读写。这种存储单元的电路如图 3-5 所示。它由一个晶体管 T 和一个电容器 C 构成，行选通线连接在三极管的栅极，使晶体管 T 打开或者关闭。数据线上传输数据，数据从被选中的列中读出或写入。写操作时行选通置 1，

T 导通，数据以充电或者放电的形式由数 据线存入到电容 C 中，数据写入后将 T 关闭，使得数据能以电荷的形式保存在 C 中。读操作时，行选通置 1，存储在 C 上的电荷通过 T 输出到数据线上，通过读出放大器电路输出。图中 CD 是数据线上的分布电容，是我们不希望存在的。为了节省芯片的面积，存储单元中的电容器 C 的容量不能做得很大，一般都比数据线上的分布电容 CD 小。因为每次读操作之后，存储的内容就受到 CD 的影响，所以在电路上必须采取措施，以再生原来存储的信息。通常是每次访问前将 CD 放电，并将读出的数据放大到规定的电平后再输出。

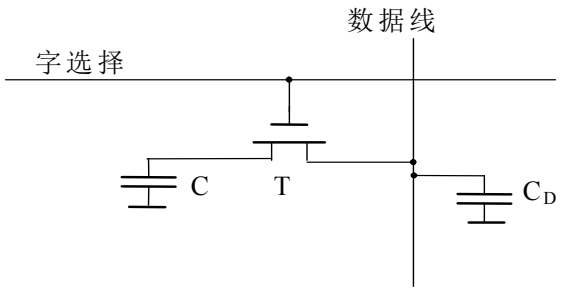


图 3-5 静态存储器的读写周期

这种存储器的特点是可以用较少的晶体管构成一个存储单元，从而提高存储器芯片的存储容量，降低存储器的成本，同时,存储器的功率消耗也比较低；但是，每次访问前对 CD 的放电操作降低了存储器的工作速度。此外，存储单元将信息以电荷的形式存储在电容器上，这样存储的信息只能保持较短的时间，通常是若干毫秒。为了使信息存储更长的时间，必须不断地刷新每个存储单元中存储的信息，也就是将各存储单元中的数据读出之后，经过放大再写回到原单元中。在电容中的信息可保持的时间决定了两次刷新的间隔时间，在这段时间内，必须将存储器中所有的存储单元刷新一遍。因为这种刷新操作必须周期性地不断进行，否则存储的信息将丢失，所以这种存储器件称为动态存储器。这样构成的存储器芯片对于任何地址的访问都具有相同的访问时间，因此又称为动态随机访问存储器（DRAM）。

在 DRAM 芯片中，同样将存储单元构成一个阵列（见图 3-6）。由于存储器容量较大时地址线数量较多，所以为减少地址线数量，将地址分成两次输入芯片。先输入行地址，再输入列地址。两次地址的输入分别由芯片的地址选通信号 RAS*和 CAS*控制。其中，RAS*是行地址选通信号，CAS*是列地址选通信号，它们都是低电平有效的信号，分别用于选择存储单元阵列中的一行和一列。在输入了行地址和列地址之后，开始进行数据的读写操作。为此芯片还需要有一个控制读或写的信号为了控制数据输出，存储器芯片还可以有一个输出许可信号。这些信号需要由存储器控制电路产生。

DRAM 的刷新操作一般也在存储器控制电路的控制下进行。通常在对存储单元进行读写操作时，是将一行的数据读出，经过信号放大后再写回，对该行存储单元的数据都进行了一次刷新操作。但为了保证所有单元中的数据都能得到刷新，存储器控制电路需要对刷新操作进行专门的控制，使得芯片不断地依次对每一行单元进行刷新。在现在的动态存储器产品中，刷新控制电路都包括在存储器芯片中，外部只需给出启动刷新操作的控制信号。刷新地址是一个行地址，可以由一个计数器提供。

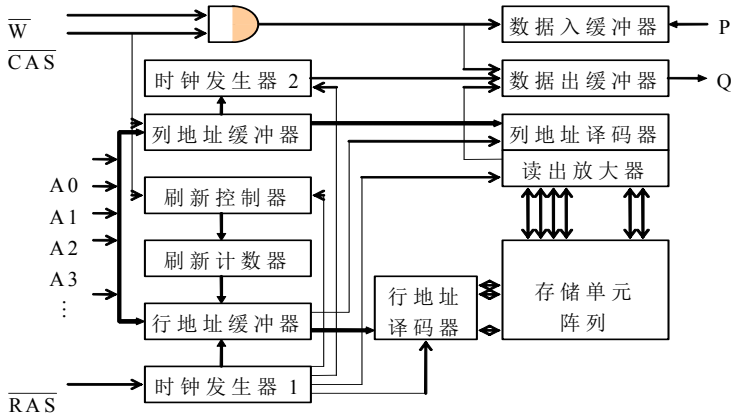


图 3-6 动态存储芯片阵列结构

在 DRAM 存储器芯片中，除了与 SRAM 相同的信号线以外，还增加了 RAS*和 CAS*信号，分别用于控制行地址和列地址的输入。动态存储器的典型读写工作时序如图 3-7 所示。在访问存储器时，各位地址信息是同时提供的。在采用动态存储器芯片的存储器中，一般需要一个控制电路，以生成一些存储器操作所需的控制信号，并且将地址信息分成行地址和列地址。在外部电路提供了行地址之后，存储控制电路向存储器发出 RAS*信号（RAS*信号下降），将行地址输入到芯片内。这时外部电路可向动态存储器芯片提供列地址，在外部电路提供了列地址之后，由存储控制电路向存储器发出 CAS*信号，将列地址输入到芯片内。这时就启动了芯片内部的读写操作，根据 CAS*信号有效时界的电平状态，决定是进行读操作还是写操作。

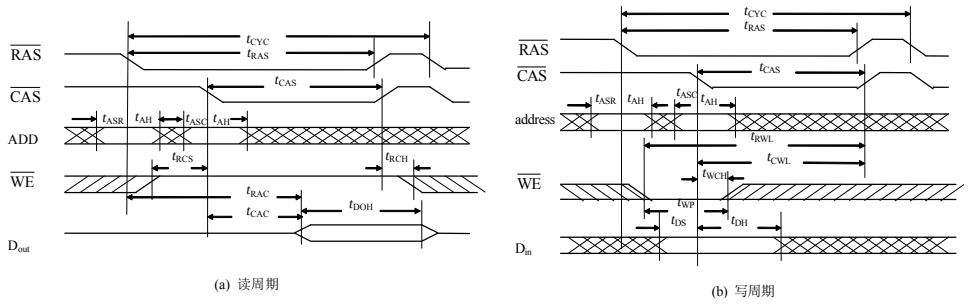


图 3-7 动态存储器的读写周期

在读访问中，在 RAS*有效之后经过访问时间 tRAS 并且在 CAS*有效后的延迟之后，数据将从数据线上输出，读出的数据在和撤销后即消失。在写访问的情况下，外部电路提供的写入数据通过 RAS*或者 WE*信号输入到动态存储器芯片中，在数据写入芯片之后，RAS*和 CAS*信号都可撤销以结束访问过程。但 RAS*和 CAS*的有效时间必须保持一定的长度，而且在 RAS*和 CAS*撤销到下一次有效之间也必须经过一段时间，而不管它是读操作还是写操作。因此在设计动态存储器电路时应当根据具体存储器芯片的数据手册进行。

动态存储器的刷新方式有多种，下面分别介绍：

(1)只用 RAS*和 CAS*的刷新。如果在 RAS*有效之后，CAS*保持不变，则动态存储器芯片就对输入的行地址所指定的行进行一次刷新操作。这时芯片的其他控制信号都是任意的，但需要外部提供刷新地址，这个刷新地址可通过一个刷新计数器提供，其时序图如图 3-8(a)

所示。

(2)CAS*在 RAS*之前的刷新。上述读写操作都是先使 RAS*有效，然后 CAS*有效，如果 RAS*在 CAS*之前有效，则动态存储器芯片将进人刷新操作过程。采用这种刷新方式的动态存储器芯片一般具有刷新地址计数的功能，不需要外部电路提供刷新地址，其时序图如图 3-8(b)所示。

(3)隐含式刷新。即在读写访问周期之内，在 RAS*信号线上加一个脉冲表示刷新命令，芯片在这信号的控制下进行刷新操作，刷新的地址也由内部计数器提供。这种刷新方式不需要用一个专门的操作周期对存储器进行刷新，可提高存储器的工作速度，是较新的刷新方式，这种刷新方式的时序图如图 3-8(c)所示。

DRAM 的主要性能参数有：

t_{RAS} ：RAS*脉冲宽度。

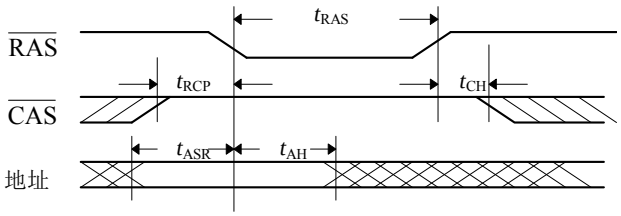
t_{CAS}：CAS*脉冲宽度。

t_{CYC}：访问周期时间。

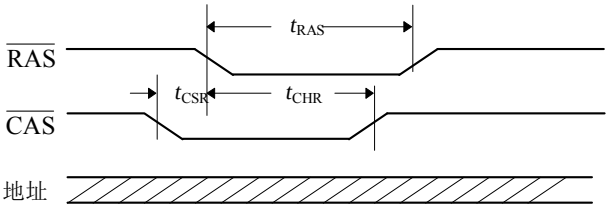
t_{RAC}：从 RAS*的访问时间。

t_{CAC}：从 CAS*的访问时间。

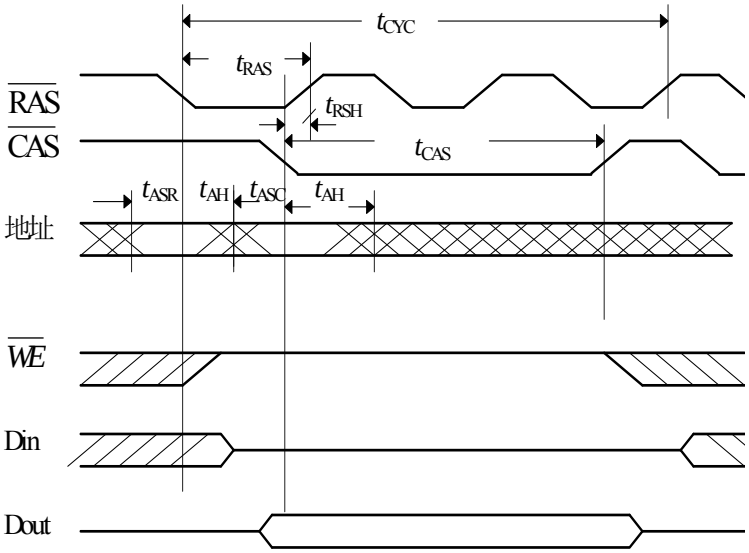
存储器控制电路生成一些存储器操作所需的控制信号。在动态存储器中，为了将行地址和列地址分别输入到存储器芯片中，存储器的控制电路需要控制地址线的选择。此外，还需要通过外部电路将 CPU 提供的地址寄存起来，并分成两步提供给动态存储器 芯片。寄存的行地址和列地址可通过一个二选一的地址多路选择开关电路进行选择。这一操作过程需要在一个存储控制电路的控制下进行。地址信号应当在这两个选通信号之前到达存储器芯片，并在选通信号下降之后保持一段时间。读写信号也应在 CAS*之前有效，并在 CAS*之后撤销。CAS*信号应滞后于 RAS*信号，在 RAS*信号的下降之前，CAS*应保持一段高电平时间。在写数据时，数据应当在 CAS*之前就有效，并保持一段时间。



(a) 只用 RAS*的刷新



(b) CAS*在 RAS*之前的刷新



(c) 隐含式刷新

图 3-8　动态存储器的刷新周期

当 CPU 信号的负载数量较多时，在存储器控制电路中还需要有数据驱动电路。数据驱动电路对数据信号进行电流放大和整形，它是一个双向驱动电路，需要有一个控制驱动方向的信号。控制电路还要负责产生读写控制信号时序和刷新控制信号时序。这样，存储器控制电路输出的控制信号包括数据的驱动控制和多路转接控制信号、地址选通信号、列地址选通和行地址选通信号等，如图 3-9 所示。其中用一个地址二选一电路，从 CPU 提供的地址中选择出行地址和列地址。一般将地址的高位部分作为行地址，低位部分作为列地址。

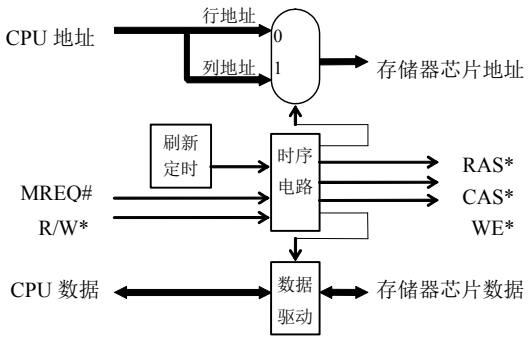


图 3-9 动态存储器控制电路

控制动态存储器的刷新时刻是控制器设计的一个重要方面。目前，动态存储器在刷新间

隔时间内安排存储阵列中每一行数据刷新时刻的方式一般采用异步方式。异步式刷新方式是将刷新操作均匀地分布在刷新间隔时间内，用一个定时器发出刷新请求，存储器的控制器根据请求在访存操作的间隙进行刷新，这样可减少集中式刷新操作对访问存储器操作的影响。近年来的动态存储器都支持隐含式刷新，在这种刷新方式下，存储单元的刷新操作可以与读写访问操作重叠进行，而不需要用专门的存储器操作周期，这样就消除了因刷新操作对访存操作带来的时间延迟。为提高刷新操作的速度，在设计字位扩展的存储器系统时，应使系统中所有存储芯片的刷新操作同时进行，而不是一个芯片一个芯片串行地进行刷新。

动态存储器控制电路是 CPU 与动态存储器的接口电路。动态存储器的控制电路已经有一些专用的集成电路芯片，如 Intel 82840 是奔腾 4 与图形控制器和主存之间的接口芯片，能够控制存储器访问的操作。动态存储器的控制电路也可以根据具体情况专门设计，一般可采用可编程芯片 PLD 或者专用芯片实现。有了动态存储器的控制电路，对于 CPU 而言，访问动态存储器就可以与访问静态存储器一样进行，只是访问的速度不同。

近十几年来，动态存储器芯片上出现了一些新的技术。在动态存储器芯片中采用的高速存取方式，如快速页访问方式、增强数据输出方式和同步访问方式等，这些访问方式可以显著提高基本 DRAM 的访问速度。

（1）快速页式动态存储器。快速页式动态存储器（FPM DRAM）中的页式访问是一种提高存储器访问速度的重要措施。在具有快速页访问方式的存储器芯片中，如果前后顺序访问的存储单元处于存储单元阵列的同一行（称为页）中，就不需要重复地向存储器输人行地址，而只需输入新的列地址即可。也就是说，存储器的下一次访问可以利用上一次访问的行地址。这样就可以减少两次输人地址带来的访问延迟。在页访问方式下，只要在输入了行地址之后保持 RAS*信号不变，采用 CAS*输入不同的列地址就可以对一行中的不同数据进行快速连续的访问。其访问速度比一般访问方式提高 2～3 倍。

快速页访问方式把一行数据锁存在读出放大器内，只选取新的列地址，从而提高数据访问的速度。如果在数据放大电路之后采用一个附加的锁存器，使得芯片可提前启动下一个页内的访存操作，则构成了增强数据输出的动态存储器（EDO DRAM）。如果再增加数据缓存的容量，存放更多行的数据，则构成了带缓存的 DRAM，称为 EDRAM 或者虚通道 DRAM（VCRAM）。可以缓存多个页的数据，可加快同时多个页的数据的交替访问。

（2）同步型动态存储器。过去的 RAM一直是异步控制的，CPU 向存储器发出地 扯和控制信号，经过一段时间把数据写入存储器或者把数据从存储器中读出。在访问延迟时间内，DRAM 完成多种内部操作过程，如输入地址、读出数据等。在这段时间内 CPU 只是等待，直到存储器响应，这影响了的执行速度。随着工作频率的迅速提高，这种影响越来越大。为了减少的访存等待时间，出现了同步型动态存储器芯片（SDRAM）。

SDRAM 存储器芯片在系统时钟控制下进行数据的读出和写入。芯片把给出的地址和数据锁存在一组锁存器中，锁存器存储地址、数据和 SDRAM 输入端的控制信号，直到指定的时钟周期数后响应。使用 SDRAM 可实现 CPU 的无等待状态。由于 CPU 知道 SDRAM 要用多少时钟周期才能响应，在 SDRAM 执行 CPU 的访问请求时，CPU 可执行其他任务。例如，一个在输入地址后有 60ns 读出延迟的 DRAM，如果 DRAM 是异步工作的，则要 CPU 等待 60ns；但是如果 DRAM 与 CPU 是同步的，在周期为 10ns 的时钟控制下工作，则 CPU 可把地址放入 SDRAM 的锁存器中，在存储器进行读操作期间去完成其他操作。然后，当计时到 6 个时钟周期以后，它所要的数据已经从存储器中读出，CPU 可以得到访问的数据。一般在的访问延迟时间内，可以传输下几个访问的地址，从而可以将几个存储器访问的操作

相互重叠，以提高存储器的访问速率。这样，在 SDRAM 中把地址的输入和相应的数据传输分开，称为分离事务技术。这里的事务（Transaction）是指地址传输、数据传输等总线上的操作。

SDRAM 芯片的内部结构如图 3-10 所示。在芯片内部，SDRAM 包含多个存储阵列，这些存储阵列轮流工作，从而缩短了存储器周期时间，在每个阵列本身的速度保持不变时，整个存储器芯片的工作速率可以提高。多个存储阵列使得 SDRAM 的访问操作可以重叠进行，当芯片在输入完前一个地址后，就可以输入提供的下一个地址，对另一个阵列中的信息进行访问。CPU 在开始下一个存储器操作之前无需等待一个访问周期。此外，在较新的 SDRAM 芯片中还采用双倍数据速率（DDR）和 4 倍数据速率（DDR2)技术，在每个时钟周期内传输 2 个或 4 个字的数据，从而提高存储器的访问速率。DDR SDRAM 中采用了双向数据选通信号，由发送方驱动，用于接收方获得准确的数据采样时间信息，从而提高数据传输速率。DDR SDRAM 中采用了延迟锁相环（DLL)技术来对齐一些控制信号，以克服信号延迟带来的不利影响。但是在 DDR SDRAM 中，尽管数据传输速率提高了 1 倍，但地址和命令的传输速率没有提高，所以总体工作速度并没有提高 1 倍，只有在连续猝发传输时的访问速度才有明显的提高。

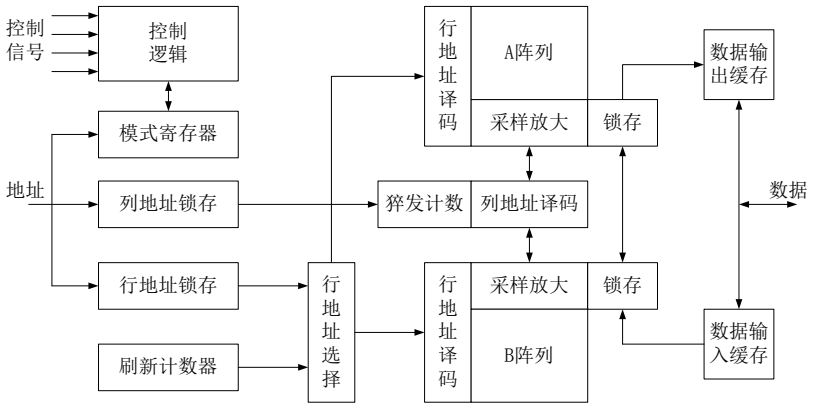


图 3-10 SDRAM 的内部结构

(3)Rambus 存储器。Rambus 公司的动态存储器（RDRAM)采用专门的存储单元阵列和高性能的芯片接口。它采用异步的面向块传输的协议传送地址信息和数据信息，并采用新型数据传输信号技术，使得数据传输的速率可提高到以上。

RDRAM 芯片采用的新的接口称为 RSL，这种接口以消息（数据包）为传输机制，使得单个芯片就像一个存储系统，而不是一个存储器芯片。RDRAM 的消息传输接口使得 CPU 可以将更多的访存操作重叠进行。RDRAM 芯片能够在一个访问请求下返回多个数据，并能对自身进行刷新。提供了字节宽度的接口和一个时钟信号， 使其能够与保持严格时钟一致性。RDRAM 允许每字节增加一位，这样典型的“Rambus 字节”是 9 位的。第 9 个位可用于奇偶校验、重叠、硬件标志等。

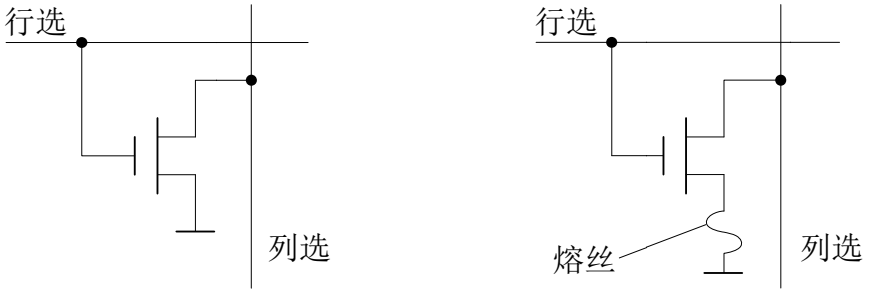
在一些改进型的 RDRAM 芯片中采用并行技术，可以使少量的随机数据访问也能获得很高的带宽，以及采用更宽的接口和更高的工作频率。在 RDRAM 中出现的另一种接口标准是双数据总线结构，支持这种接口的 RDRAM 称为 Direct RDRAM。它增加了数据线的宽度，并提高了频率，使得总线协议更加有效。这种新的信号传输技术进一步提高了数据传输速率，现在已经能够支持的操作速度。2000 年, RAMBUS 提出了它的新一代信号传输技术 QRSL，在每条线路上提供双倍的数据速率。 RDRAM 存储器的高速接口使得它能够支持高

速的访存带宽需求。

3.1.3. ROM 的结构和原理

上述 SRAM 和 DRAM 的共同特点是，当电源撤销时，存储的数据也随之丢失，因此称为挥发性（Volatile）存储器。磁性介质和光介质存储器具有非挥发性的特点，能够在关机的情况下保存数据，因此成为计算机存储器的一个重要方面，如磁盘和光盘等。在半导体存储器中，只读存储器 ROM（Read Only Memory）是非挥发性的存储器。

ROM 是一种只能读取数据不能写入数据的存储器。它用于存储计算机中的一些固定信息，如计算机的启动程序。其中存储的数据在芯片的生产过程中写入，在计算机系统的工作过程中不能进行修改。MOS 型 ROM 存储单元的电路结构也与动态存储单元的电路结构类似，用一个晶体管构成一个存储单元。存储单元构成阵列，用行选通和列选通信号选择存储单元。这类 ROM 的存储单元有各种构成方式，可以用半导体二极管、双极型三极管和 MOS 三极管电路构成。图 3-11 (a)所示是用 MOS 晶体管构成存表储 0 单元的单元电路，当单元中不包含晶体管时构成 1 单元。在这种电路中，当一行被选通时，相应的行选通信号使得该行上的各晶体管导通。当晶体管的源极接地，漏极与相应的列选通信号线连接时，晶体管的导通使得该列选通信号线输出低电平。没有晶体管的单元将通过连接正电源的电阻使得列信号线为高电平。这种结构的 ROM 称为掩膜式 ROM，因为晶体管的存在与否由芯片的生产过程中的一个称为掩膜的绝缘材料。



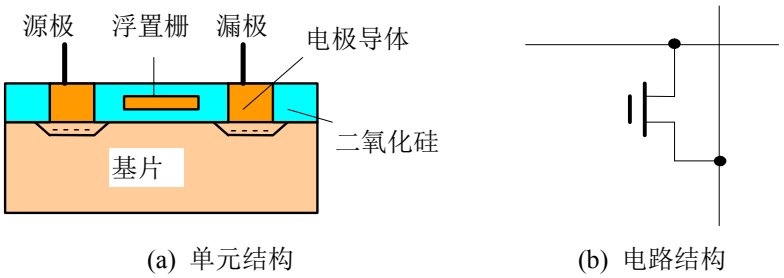
（a）掩膜式ROM的0单元　　（b）PROM的单元

图 3-11 ROM 存储单元的例子

为了使用户能够写入自己的数据，可以采用可编程的 ROM，即 PROM。PROM 通过在晶体管的发射极与列选通线之间用熔丝进行连接，从而实现可编程的数据存储，如图 3-11（b）所示。在初始情况下，各存储单元的晶体管都导通，输出的数据内容都是 0。如果将熔丝烧断，则晶体管不能导通，输出的数据为 1。用户可用写入设备通过较高电压将某些存储单元的熔丝烧断，从而在这些单元中写入 1。这种通过烧断熔丝将数据写入芯片的过程称为对 ROM 进行编程。由于熔丝烧断后不可恢复，所以 PROM 只能被用户编程一次，以后就不能修改存储的数据，通常称为 OTP（One Time Programable）。

另外一种可编程的 ROM 允许存储的内容被擦除以写人新的数据，它采用可编程的存储单元，将已写入数据的存储器芯片放在紫外线下照射一定时间就可擦除其中的内容，这种芯片称为可擦写 PROM，或 EPROM。这种 ROM 的存储单元一般采用 MOS 晶体管，其栅极是一个被绝缘体隔绝的浮空的多晶硅，称为浮置栅极（Floating Gate）， 如图 3-12 所示。初始时浮置栅极上没有电荷，因而晶体管不导通。编程时在较高电压的作用下，可向浮置栅极注入电荷，使晶体管导通。由于浮置栅极被绝缘隔绝，其电荷不致流失，所以晶体管能够一

直保持导通或者截止，从而存储数据。为了擦除数据，这种 EPROM 芯片上有一个石英玻璃窗口，当紫外线照射这个窗口时，所有存储单元中的浮置栅极上的电荷在紫外线的激发下会形成光电流泄漏掉。数据被擦除后就可以重新编程。这种可擦写的编程方式使得硬件的开发可以在开发的现场实现芯片的编程，是一种现场编程。



(a) 单元结构　　(b) 电路结构

图 3-12 EPROM 存储单元结构

上述只读存储器的数据不能擦除或者必须采用专门设备进行擦除或编程。为了擦除数据或者进行编程，必须将存储器芯片从系统中拔下。许多应用场合需要在线更新只读存储器中的数据，为此开发出了一种能够用电子的方法擦除其中的内容的 EPROM 存储器芯片，称为电可擦写 PROM（EEPROM 或 E2PROM）。EEPROM 是一种可用电子方式对存储芯片中的某个存储单元进行擦除以及编程的非挥发性存储器件。但是，EEPROM 的单元结构较为复杂，芯片的成本较高，因此其应用范围受到限制。与 PROM 一样，EEPROM 的擦写次数不是无限的，但一般可达 1 万次以上。

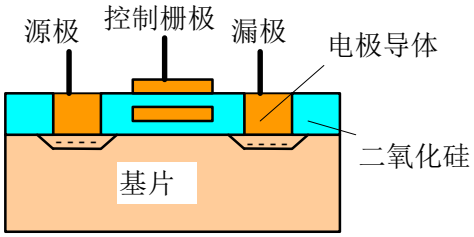


图 3-13 EEPROM 存储单元结构

EEPROM 的擦除机理与 EPROM 的机理不同，在 EPROM 线激发浮置栅上的电荷以达到擦除的目的。EEPROM 增加了一个控制栅极，如图 3-13 所示。在擦写数据时，较高的编程电压 VPP 施加在源极上，控制栅极接地，在此电场的作用下，浮置栅上的电子就越过氧化层进入源区，流入外加的电源。数据的擦写过程分为擦除操作和写入操作两步，也就是说，在写入数据之前需先将原有数据擦除，使各存储单元都处于 1 状态，然后通过写操作将数据写入。擦除操作有字节擦除和全片擦除两种，由写操作允许信号 WE*的宽度进行控制。在字节擦除时，的脉冲宽度为 10ms；如果 WE*脉冲的宽度达到 20ms，则进行全片擦除。擦除时需要采用 21V 电压，为此而设置的升压电路一般集成在芯片内，外部只提供单一的 5V 电源。EEPROM 含 2 个晶体管。电可擦写的编程方式不仅可以实现现场编程，而且还可以实现芯片在工作电路中的可编程。这种编程方式称为在线编程，具有这种特点的可编程芯片称为在线可编程芯片。

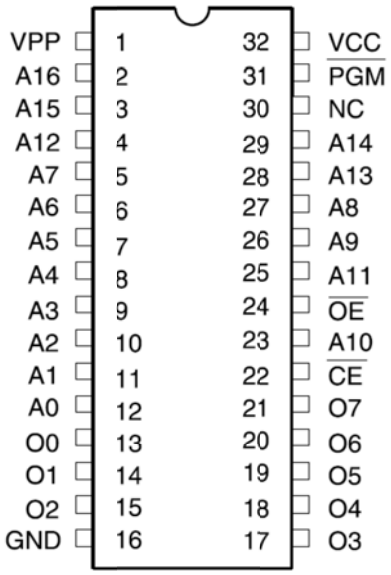


图 3-14 ROM 芯片的电路图表示

在电路图表示中，上述各种 ROM 芯片都与 SRAM 十分相似，只是没有写操作许可信号线。芯片的读操作过程也与 SRAM 的操作过程十分相似。图 3-14 所示的是 ROM 电路图符号表示，其中增加了一条控制编程的信号线 $\overline{\text{PGM}}$ 。

另一种非挥发性存储器是近年发展起来的快闪存储器（Flash memory），又称快擦存储器，它是在 EPROM 和 EEPROM 的制造技术基础上发展起来的一种新型的电可擦除非挥发性存储器元件。它的存储单元结构与 EEPROM 差别是栅极氧化层厚度不同。快闪存储单元的氧化层较薄，使其具有更好的电可擦性能。快闪存储器的擦除、重写的速度比 EEPROM 快，初期的快闪存储器像 EPROM一样只能进行全片的擦除，不能擦除一个字节。新由快闪则可以擦除一块数据，因而更适于存储文件方面的应用。现在快闪存储器的擦写次数在百万次以上。快闪存储器的存储单元包含 1 个晶体管，与 EEPROM 相比，具有密度高、价格低、可靠性高的优点。

快闪存储器的操作包括读操作和擦写操作两种 6 读操作的过程与其他存储器芯片的基本相同，外部只要提供地址、读操作信号和片选信号，存储器就输出保存的数据。存储单元在写入数据之前必须先擦除。如果旧的数据没有被擦除，写操作的结果会是新旧数据的逻辑组合。

根据存储单元的逻辑门电路特征，快闪存储器芯片可以分为 NOR 型和 NAND 型两种。在 NOR 型快闪存储器芯片中，每个单元的晶体管与标准的场效应管类似，只是具有两个栅极。NOR 型快闪存储器芯片的读操作与一般 RAM 的类似，可以随机地对任意地址进行读操作。读访问时间可以小于 90ns，适合于存储程序代码，如存储计算机启动时的代码 BIOS，以及作为 CF 存储卡中的存储器件。编程时，电流从源极流向漏极，擦除数据时在源极与控制栅极之间施加较强的电压，将浮置栅极上的电荷吸出，这些特点与 EEPROM 的类似，编程时间在 200μs 左右。NOR 型快闪存储器芯片的擦 除比较复杂，是一个比较慢的过程，与制造商的工艺技术有关，通常需要在软件的控制下进行，块擦除时间约需 0.5s。控制快闪存储器擦除过程的软件是一个驱动程序。这种驱动程序为上层软件提供一个标准的接口，使得不同产商的存储器擦除操作的细节被屏蔽。

NAND 型快闪存储器芯片采用串行的晶体管连接结构，连接结构就像与 CMOS 非门电路中的晶体管一样，每个存储单元中包括多个晶体管，构成多个存储位。这种电路结构使得晶体管之间的间隔可以做得更小，芯片的集成度更高。这种快闪存储器具有较快的擦写速度和较低的存储成本，允许的擦写次数更多，但是数据的访问是顺序方式的。数据的读写操作以一个数据单元（称为页）为单位进行，通常一页中包含 512B 数据加上 16B 校验位。数据的擦除则以块为单位，每个数据含若干个页（典型的是 32 页）。NAND 型快闪存储器芯片的页读取时间在 60μs 以上，页编程时间在 200μs 左右，块擦除时间约需 2ms。NAND 快闪存储器的数据传输方与磁盘类似，其输入/输出线是地址线与数据线公用的。一般的闪存的接口线 8 条，每条数据线每次传输 512 +16 位的数据，8 条线传输一页的数据。也有一些 NAND 闪存采用 2KB 的页，或者采用 16 位接口线。快闪存储器可用于在某应用中代替磁盘。如目前广泛使用的 SM，MMC，SD 等存储卡和 U 盘就是采 NAND 型快闪存储器的便携存储设备。快闪存储器还广泛应用于手机、数码相机、手持式计算机等嵌入式计算机系统中。

快闪存储器作为数据存储时具有高达一百万小时的平均无故障时间，而且工作速度比硬盘高，功耗低，体积小。快闪存储器还可用于代替 ROM，即用于作为系统软件核心部分的存储器，作为系统参数的存储器，以及用于作为各种逻辑电路元件等。此外，快闪存储器还可用于计算机其他外围设备中进行数据采集。由于快闪存储器具有整体电擦除和按块重新编程的功能，所以很适合于数据采集系统，进行各种数据记录。

电可擦写的只读存储器虽然可以反复修改数据内容，但是相对于而言，它的擦写速度慢、擦写操作复杂，所以不能当成随机访问存储器使用。可作为随机访问存储器使用的一种非挥发性存储器产品是在静态存储器芯片中集成可充电电池的芯片，这种存储器芯片的工作方式与 SRAM 芯片一样，而且在电源关闭后可在较长时间内保持存储的数据不丢失。这种存储器芯片称为 NVRAM(Non-Volatile)。

3.2. 存储系统的构成

存储器与中央处理器的连接包括地址线的连接、数据线的连接和控制线的连接。在访问存储器的过程中，CPU 要向存储器提供地址信息。提供的地址信号通常表示为 A0、A1、A2 等，其中 A0 通常表示最低位地址；数据线通常表示为 D0、D1、D2 等，D0 表示最低位地址。地址线是单向的，数据线是双向的传输线。读写控制信号线如 R/W*、它在高电平时表示读操作，低电平时表示写操作。控制存储器操作的信号通常还有访存请求信号，可表示为 MREQ，它在 CPU 访存期间有效。CPU 的这些引脚可以直接与 SRAM 芯片连接，CPU 的地址线可以直接与 SRAM 的地址线连接，CPU 的数据线可以直接与 SRAM 的数据线连接，CPU 的控制信号有时可以直接连接到 SRAM，有时则需要经过逻辑门电路的转换。在连接 CPU 的地址线和数据线时，需要考虑引脚上的信号工作速度是否符合存储器的工作速度要求。 产品和存储器芯片产品都有不同工作速度的版本，需要选择适当的产品型号。CPU 的地址线和控制线通常不能直接与 DRAM 直接连接，需要有外部接口电路进行转换。

CPU 在读取 SRAM 数据的过程中，首先驱动地址线，将数据的二进制地址送到存储器芯片。然后 CPU 将 WE*控制信号置高电平，表示读操作；同时将 CS*信号和 OE*信号置低电平，使得存储器芯片进行操作。存储器芯片将数据从存储阵列中读出后驱动数据输出线，将存储的数据输出。这时，CPU 就可以采集数据了。CPU 在向 SRAM 写入数据的过程中，首先也是驱动地址线，将数据的二进制地址送到存储器芯片；然后，CPU 驱动数据线，将需要写入的数据送往存储器芯片；此外 CPU 还将控制信号 CS*和信号置低电平，OE*信号置高电平，使得数据能够送入存储器芯片，并使存储器芯片进行写操作；经过一定的延迟之

后，数据线上的数据信号就写入到地址线信号所指定的存储位置中。

通常，一个存储器芯片不能满足计算机存储器的字数要求和数据宽度的要求。一个存储器一般需要许多存储器芯片构成，需要采用各种容量扩展的技术构成所需的主存储器。用若干存储器芯片构成一个存储器系统的方法主要有位扩展法、字扩展法和字位扩展法。

3.2.1. 位扩展

位扩展法用于增加存储器的数据位，如图 3-15 所示。图中，A2～A12 表示地址线 A2，A3，…,A12，共 11 条；D0～D31 表示数据线 D0，D1，…，D31，共 32 条。下标表示信号线的编号。在结构框图中，一般我们用波浪线表示同一类型的一系列编号连续的信号线。在 EDA 软件中则通常表示为数组，如 D[0..31]，A[0..31]。

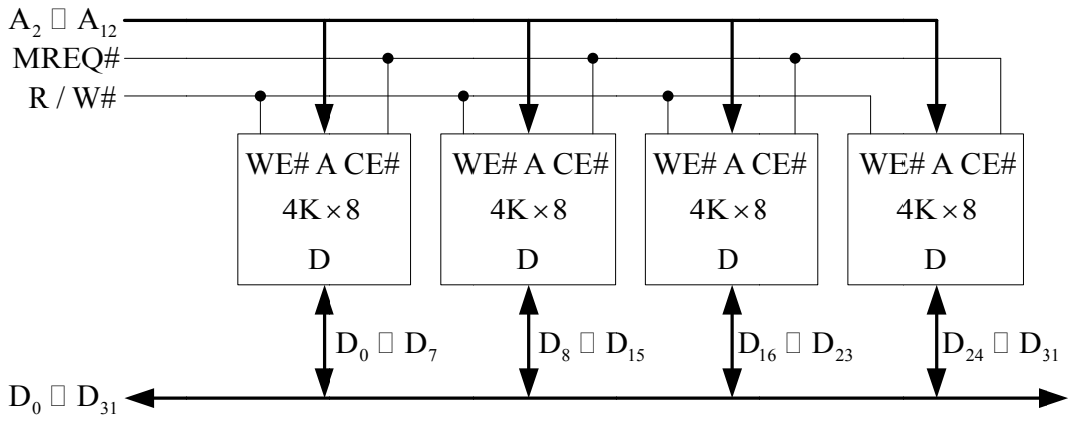


图 3-15 用位扩展方式构成的存储器

在位扩展法中，各存储器芯片采用相同的地址信号，各存储器芯片的数据线分别连接到 CPU 的数据总线上的相应位。每个存储器芯片实现整个存储器中的若干位 段。CPU 的读写控制信号 R/W#可以直接连接到存储器芯片的 WE#端，CPU 产生的 MREQ 信号可以直接连接到各存储器芯片的 CE#端。在这种方式中，所有的芯片在同一个控制器的控制下同时进行相同的操作，整个存储器的字数与每个存储器芯片的字数是一样的，位数是各存储器芯片数据位数之和。在这种方式下，地址线的负载数为存储器芯片数，数据线的负载数为 1。存储器的位数一般扩展到的字长位数。

图 3-16 所示的存储器是将 4 片 2K×8 位的存储器芯片构成一个 2K×32 位的存储器。这样构成的存储器访问操作只能以 32 位的字为单位进行。

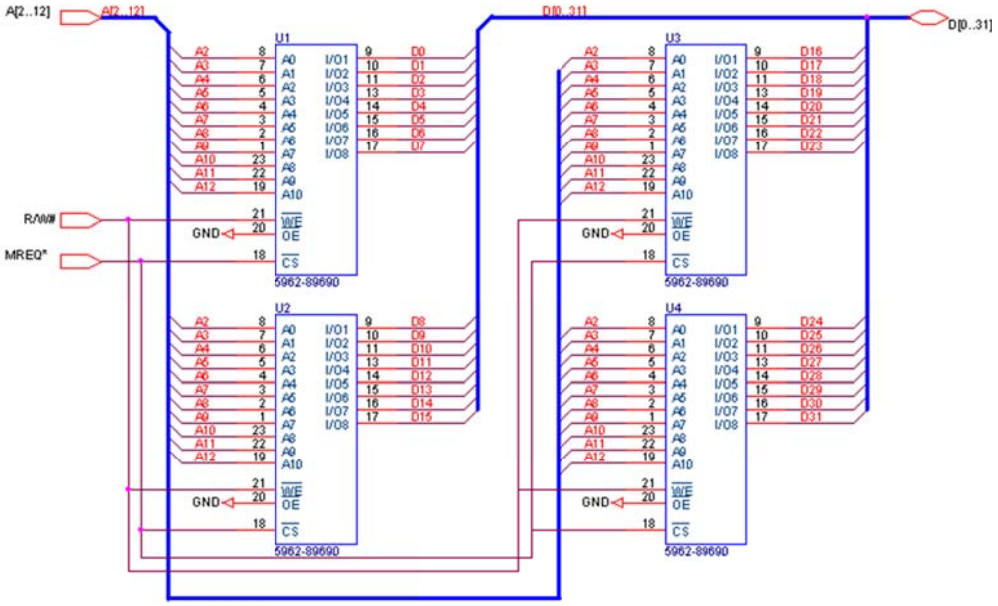


图 3-16 存储器芯片的位扩展连接

CPU 访存时提供的地址是字节地址。字节地址对存储器的每一个字节进行编址， 它的基本地址单位是一个字节。如果访问的是一个数据字，那么 CPU 实际上需要访问的是若干个连续的字节。从硬件角度，存储器地址可以有字节地址和字地址两种。字地址对存储器的每个字进行编址，它的编址单位是一个字。如果存储器字长为 16 位，那么字地址就无法区分一个字中的两个字节，存储器的访问是对同一字地址的两个字节的访问。字长为 16 位时，字地址比字节地址少一位，因为存储器中包含的字数是字节数的一半。将字节地址的最低位去掉就得到了该字节所在的字地址。去掉的这一位地址用于区分一个字中的两个字节，称为字内地址。

如果存储器以字方式组织，访问存储器就必须以字为单位，一次读写一个字。例如，对于图 3-16 所示的位扩展的存储器，每个存储器芯片都在相同的控制线的控制下同时进行读写操作。以 32 位的字为单位进行操作，CPU 每次访存都是以 4 个字节为单位进行。因为存储器是 32 位的结构，包含了 4 个字节，每次访存读写同一个字的 4 个字节的数据，占 4 个字节的地址，不管 A1 和 A0 是什么，都访问这 4 个字节，所以在访存中 A1 和 A0 不起作用，CPU 不需要向存储器提供 A1 和 A0。类似地，在 16 位结构的存储器中，A0 不起作用。如果存储器能够访问一个字中的不同字节，那么就需要字内地址（A1 和 A0）对字节进行选择，从字中选择一个字节。在有些 CPU 中，为了访问一个字节，采用字节选择信号的方法。例如，用信号 BE1 和 BE0 连接到芯片的 CS 端， 以分别选择一个字中的高字节和低字节。当 BE1 和 BE0 都有效时表示访问一个字。在 32 位的存储器中，用字节选择的方法需要 4 条字节选择信号线。

在计算机中，CPU 需要提供的地址位数取决于整个存储器的总容量（字节数）。每个存储器芯片需要的地址线数量取决于芯片的字数。对于图 3-16 所示的存储器， 2K×8 位的芯片需要 11 条地址线（211=2K），为 A10～A0。这里，整个存储器由 4 片 2K×8 的芯片构成，位扩展后是 2K×32 位的结构，总容量是 8KB，所以 CPU 提供的地址线有 13 条（A12～A0），其中 A12 连接到芯片的 A10、A11 连接到芯片的 A9，…，A2 连接到芯片的 A0。CPU 的低位地址线 A1 和 A0 不连接到存储器，因为这个存储器只能访问一个字的数据，不能只访问

某一个字节。

3.2.2. 字扩展

字扩展法用于增加存储器的字数。它在字的方向上进行扩充，而位数不变，如图 3-17 所示。各存储器芯片的同一位数据线相互连接起来，并与 CPU 的对应数据线连接，整个存储器的位数等于每个存储器芯片的位数。字扩展扩大了存储器的空间，各存储器芯片分别实现整个存储器空间的某一个区间。访问的数据位于其中某一个存储器芯片中，所以每次访存时只有一个存储器芯片进行读写操作。在连接和存储器时，将的地址分成两部分，低位部分直接送到各存储器芯片，高位部分经过译码后送到存储器的片选输入端 CE#。地址的高位部分经过译码器后，产生的每一条输出信号有效时代表访问某一段存储器的地址范围，选中某一个存储器芯片。在 图 3-17 中，将 4 片 2K×8 位的芯片扩展成 8×K8 位的存储器。容量仍然是 8KB，CPU 的地址线的低 11 位直接送到各存储器芯片的对应地址线引脚，A12 和 A11 送入一个 2-4 译码器，则产生的 4 条输出信号分别代表 4 个地址区间的选择信号，这 4 个地址区间是 0～2047、2048～4095、4096～6143、6144～8191，即二进制的 0000000000000～0011111111111、0100000000000～0111111111111、1000000000000～1011111111111、1100000000000～1111111111111。这里每个区间的低 11 位地址是相同的，不同区间的地址高 2 位不同，因而高 2 位地址作为选择区间的地址码。CPU 的 MREQ 信号作为译码器的控制信号，CPU 访存时 MREQ#无效，这时译码器没有有效的输出信号，各存储器芯片都不工作。在这种连接方法中，数据线和低位部分地址线的负载数为存储器芯片的个数，高位部分地址线的负载为 1。

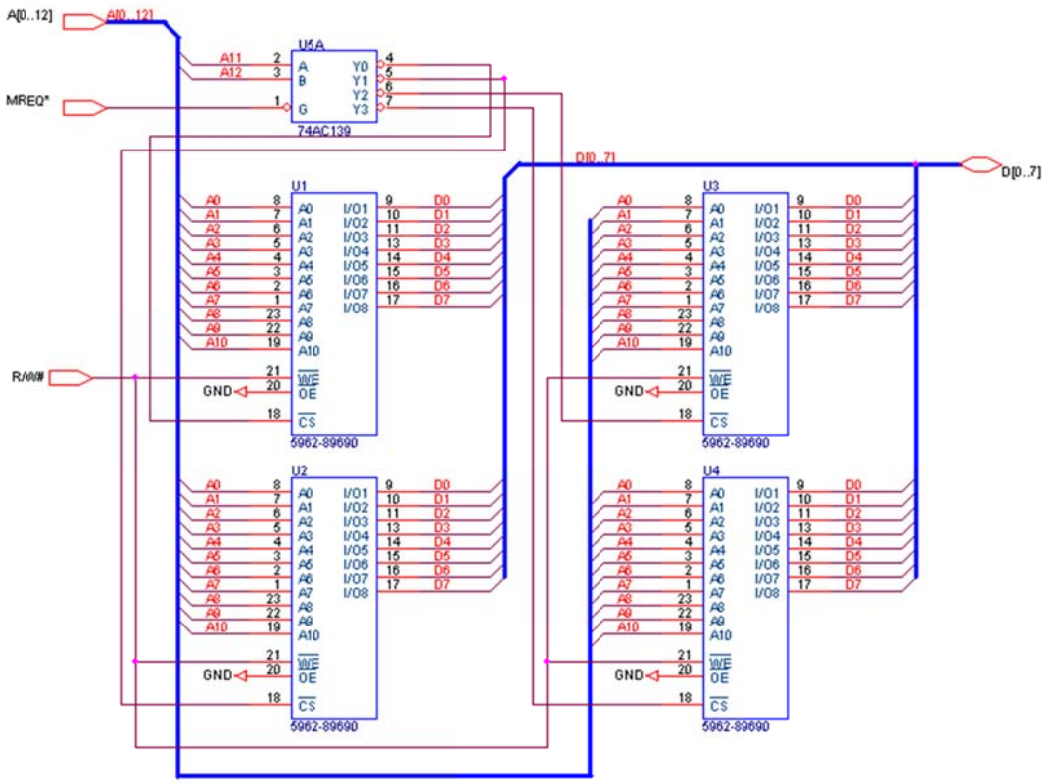


图 3-17　用字扩展方式构成的存储器

3.2.3. 字位扩展

字位扩展法则是上述两种扩展方法的组合，既在位方向进行扩展，又在字方向进行扩展。字位扩展的例子如图 3-18 所示，图中把经过上述位扩展的存储器模块作为扩展的基本模块，再通过字扩展增加存储器的容量。字位扩展法是容量较大的存储器系统中广泛采用的方法。它用一组存储器芯片的位扩展构成的模块实现存储器所需的位数，用若干组芯片的字扩展实现更大的存储容量。在字位扩展法中，数据线的负载数为扩展后的模块数，低位部分地址线的负载数为存储器芯片数，高位部分地址线的负载数为 1。 在采用内存条的计算机中，内存条之间一般构成字扩展，内存条内部是位扩展，这样就构成了一个字位扩展的存储器。存储器字位扩展法中需要的存储器芯片数量可以这样计算：如果采用的存储器芯片容量是 $m \times n$，组成的存储器容量为 M 字，每字 N 位，那么需要的芯片数为

$$\frac{M}{m} \times \frac{N}{n}$$

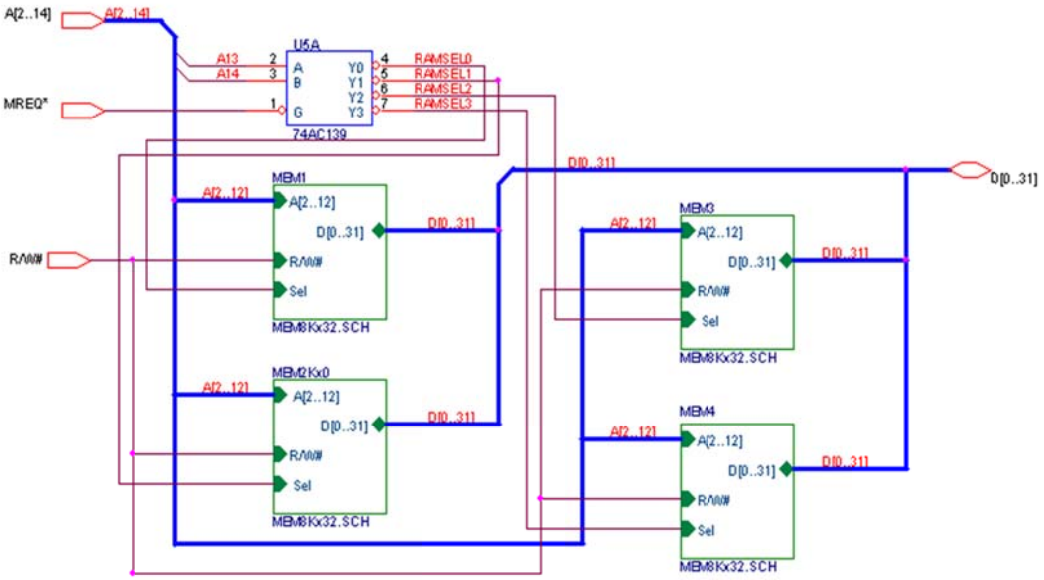


图 3-18 用字位扩展方式构成的存储器

采用字位扩展方式构造存储器时所需要芯片数量与芯片的结构以及存储器的字位结构有关。对于 32 位存储器，如果采用的存储器芯片是 1 位数据的，则至少需要 32 片。如果采用 8 位数据的芯片，则最少可以只有 4 片芯片构成 32 位的存储器。在采用内存条的计算机中，内存条之间通常是字扩展的关系，构成字位扩展方式。内存条的字扩展方式使得我们可以通过增加内存条的数量来增加存储器的容量。

在将存储器与 CPU 连接时，应当注意 CPU 的负载能力。因为 CPU 的线路负载能力有限，在存储器负载较大时应采用缓冲器或驱动器，其次要考虑时序的配合，即保持 CPU 时序与存储器速度的一致。一般 CPU 的访存信号及其时序能与静态存储器的信号匹配,动态存储器与 CPU 连接一般需要外加接口电路。接口电路中有控制器生成必要的工作信号，还有驱动门电路对数据信号线进行驱动，使其能够连接更多的存储器芯片。动态存储器还需要处理刷新的问题。此外，在字位扩展的存储器中还应考虑片选信号的安排。在采用字扩展的存储器中都需要对存储器芯片进行选片，根据地址选择不同的芯片组。

【例 3-1】设有若干片 256K×8 位的 SRAM 芯片，问如何构成 2048K×32 位的存储器？需要多少片 RAM 芯片？该存储器需要多少地址位？画出该存储器与连接的结构图，设电路的接口信号有地址信号、数据信号、控制信号 MREQ#和 R/M#。

答：芯片的字数比需要构成的存储器少，需要进行字扩展；SRAM 芯片为 8 位，比需要构成的存储器少，需要进行位扩展；所以需要采用字位扩展的方法。该存储器需要 2048K/256K×32/8 = 32 片 SRAM 芯片，其中每 4 片构成一个字的存储器芯片组，8 组芯片进行字扩展。

如果采用字寻址方式，需要 21 条地址线，其中高 3 位用于芯片选择，低 18 位作为每个存储器芯片的地址输入。因为存储器容量为 2048K×32=232KB，所以访存的最高地址位为 A22。

用 MREQ#作为译码器芯片的输出许可信号，译码器的输出作为存储器芯片的选择信号，R/M#杯作为读写控制信号，该存储器与连接的结构框图如图 3-19 所示。其中把位扩展的一组芯片画成一叠，因为它们采用相同的控制信号，相当于一个经过位扩展的子模块。另外，图中没有画出全部 8 个芯片组，而是用破折号表示了这种省略，这是在结构框图常见的表示方法，在电路图中则不允许这种省略。

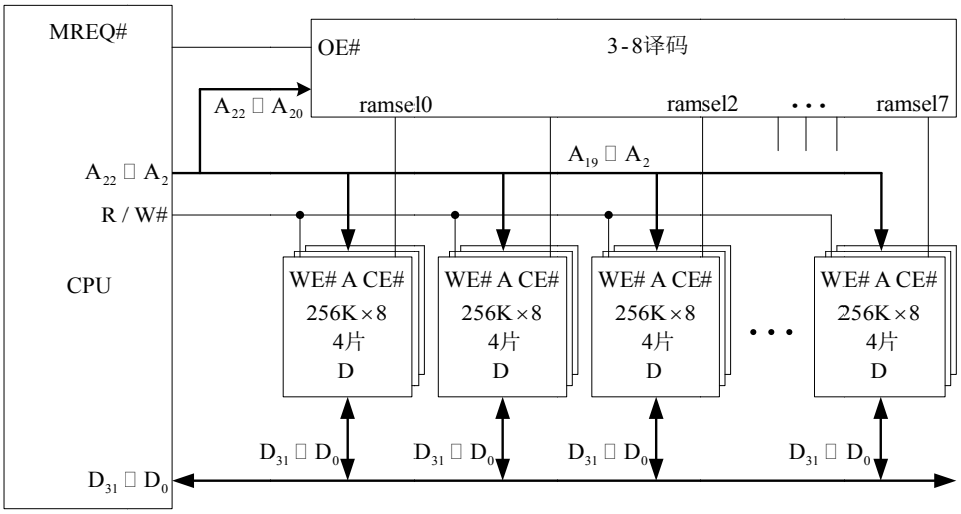


图 3-19 存储器与 CPU 的连接框图例子

3.3. 高速缓冲存储器

在实际的计算机系统中，因为的工作速度提高很快，所以对于存储器的速度和容量的要求越来越高，为此需要采用提高存储器工作速度的技术，或者采用层次化的存储器系统结构。高速缓冲存储器（Cache)就是在这种背景下产生的。本节介绍 Cache 的原理。

3.3.1. 基本原理

存储器的速度指标包括访问时间、访问周期时间和带宽。通过数据线和地址线与主存储器连接。从一次启动存储器操作到操作完成后可启动下一次操作的时间称为存储器周期时间，即访问周期时间。它是存储器芯片在连续两次启动访问存储器操作之间必须间隔的时间。从启动访问存储器的操作到获得访问数据的时间称为访问时间，即从提供地址和读操作信号到

数据读出所经历的时间。存储器的周期时间一般比访问时间长，因为在一次访问完成之后存储器需要一定的恢复时间才能开始下一个访问操作。带宽为存储器在连续访问时的数据吞吐速率。带宽的单位通常是每秒钟传送的位数或字节数。

在计算机中，CPU 的工作速度一般都大于主存储器的工作速度。主存储器一般用动态存储器实现，高速的静态存储器芯片因为容量低、价格高而不适于组成大容量的主存储器。而访问存储器是工作的基础，访问操作的速度与工作速度的匹配对计算机的工作速度具有重要的影响。因此高性能计算机的瓶颈之一是 CPU 与存储器的接口。存储器设计的目标之一是以较小的成本使存储器与口的速度相匹配，或者说达到与 CPU 相对应的工作速度和数据传输吞吐率；同时还要求存储器有尽可能大的容量。因此，速度、容量和价格是存储器设计应考虑的主要因素。

硬件系统中有这样一个普遍的规律：就是系统的复杂性越小，系统能够达到的工作速度就越高，因为简单的系统中逻辑门的延迟较小；系统的尺寸越小，系统能够达到的工作速率越高，因为尺寸小的系统中信号在线路上的传输延迟较小。对于存储器系统， 高速、大容量和低成本这三个因素是相互制约、相互矛盾的。存储器容量的增大一般意味着速度的降低，例如静态存储器的工作速度是存储器件中最高的，但将大量的静态存储芯片构成一个大容量的存储器，则不仅价格昂贵，其工作速度也将由于体积大、传输距离大和线路复杂而降低。因此，设计一个实用的存储器系统需要从结构上采用专门的技术。尽管计算机程序往往需要巨大的快速存储空间，但程序对其存储空间的访问并非是均匀分布的。从大量的统计中可以得到这样一个访存规律：程序对存储空间的 90%的访问局限于存储空间的 10%的区域中，而另外 10%的访问则分布在存储空间的其余 90%区域中。这一规律称为访存局部性规律。人们进一步发现计算机程序对存储器的访问有这样两种局部性规律：

（1）时间局部性：如果一个存储单元被访问，则可能这个存储单元会很快再次被访问。

（2）空间局部性：如果一个存储单元被访问，则它邻近的单元可能很快被访问。

形成上述规律的原因包括程序的顺序执行和程序的循环等。在程序顺序执行时，下一次执行的指令和上一次执行的指令在存储器中的位置是相邻的或相近的；在循环程序中，循环体的指令要重复执行，相应的数据要重复访问。程序和数据的存放都有一定的局部性规律而不是均匀分布的。

根据访存的局部性规律优化设计存储器系统，就是要求将计算机中频繁访问的数据存放在速度较高的存储器件中，而将不频繁访问的数据存放在速度较慢但价格较低的存储器件中，为此人们想到了层次化的存储器实现方法。存储器系统根据容量和工作速度分成若干个层次。一般速度较慢的存储器件成本较低，可用其实现大容量的较低层次的存储器，而用少量的速度较高的存储器件实现速度较高的存储器层次。在多个层次的存储器中，上一层次的存储器比其下一层次的容量小、速度快、每字节存储容量的成本更高。这样既可以用较低的成本实现大容量的存储器，又使存储器具有较高的平均访问速度。其余 90%中的寄存器可看做是最高层次的存储部件，它容量最小，速度最快，但对寄存器的访问不按存储器地址进行，而是按寄存器名进行，这是寄存器与存储器的一个区别。寄存器以下可以有高速缓存、主存、外存等层次。外存是最低层次的存储器，通常用磁盘、磁带或光盘等构成，其特点是容量大、成本低、速度慢。

在这样一个层次结构的存储系统中，Cache 的作用是弥补与主存储器在速度上的差异。在存储系统中设置外存层次的目的是扩大程序可访问的存储器空间，通过把磁盘空间当作内存空间供程序使用，我们就建立起一个虚拟存储器。这个虚拟存储器的容量可以相当于磁盘

的容量，而对于程序来说，使用虚拟存储器就像使用主存储器一样方便。这样就解决了主存容量不够的问题。在安排存储的数据时，我们把程序使用比较频繁的数据或者指令存放在主存中，而把不频繁使用的数据或者指令放在磁盘上，这样就可以使得虚拟存储器在增加存储容量的情况下，总体工作速度接近于主存的工作速度。

Cache 是一个高速小容量的临时存储器，可以用高速的静态存储器芯片实现，或者集成到 CPU 芯片内部。Cache 是一个采用算法进行管理的缓存，存储 CPU 最经常访问的指令或者操作数据。Cache 的管理算法完全由硬件电路实现，以满足工作速度的要求。由于高速的存储器芯片的价格太高，不适于实现大容量的主存储器，在主存的速度不能满足 CPU 要求的系统中采用 Cache 这一层次的存储器是提高系统性能的有效手段。Cache 最初出现在大型计算机中，但现在已被各种类型的计算机广泛采用。一些微处理器中已经将 Cache 集成在 CPU 芯片中，而且还出现了具有两级和三级 Cache 的计算机系统。

在带有 Cache 的计算机中，Cache 中开始时是没有数据或程序代码的。当 CPU 访问存储器时，从主存中读取的数据或代码在写入寄存器的同时还写入 Cache 中。在以后的访问中，如果访问的数据或代码已经存在于 Cache 中，就可以直接从 Cache 中访问到数据或代码，而不必再到主存储器中去访问了。访问主存的数据或代码存在于 Cache 中时的情形称为 Cache 命中（hit），Cache 命中的统计概率称为 Cache 的命中率。相应地，访问主存的数据或代码不存在于 Cache 中时的情形称为不命中或失效（miss），不命中的统计概率称为失效率。

为了提高 Cache 的命中率，在将主存中的数据或代码写入 Cache 时，一般把该数据前后相邻的数据或代码也一起写入 Cache。也就是说，从主存储器到&的数据传送一般是以数据块为单位进行的，这样既提高了 Cache 的命中率，又提高了数据传输的效率。因为根据访存的局部性，访问的数据集中在较小的范围内。数据块的大小一般是 2 的幂次。这样，每个数据块的起始地址也是 2 的幂次。在 Cache 命中时所需的访问时间 称为命中访问时间，不命中时因访问主存而增加的访问时间称为 Cache 的失效访问时间 (miss penalty)。在访问存储器时，如果 Cache 命中，则访问时间是 Cache 的访问时间，即命中时间；如果 Cache 失效，则访问时间是命中时间加上 Cache 的失效访问时间。在具有 Cache 的系统中，无论耽是否命中，Cache 都是必须访问的。

高速缓存的设计中要考虑的问题是：

（1）主存中的块放入 Cache 中的什么地方？这是一个将主存地址映像到 Cache 地址的问题，即地址映像方法。它确定主存地址与 Cache 址的映像关系。主存地址到 Cache 地址的映像以块为单位。

（2）Cache 放满时怎么办？这时需要有一个算法将 Cache 中的某一个块替换出去，即要有一个块的替换策略。它决定将中的哪一块数据移去以调入访问的块。

（3）写 Cache 时是否写主存？即块的更新策略。块的更新策略决定在写操作时，何时将数据写入主存，何时将数据写入 Cache。

此外，还需考虑 Cache 的容量、块的容量等。Cache 的容量一般比主存低得多，通常在 1KB～256KB 之间，与主存传输的数据块的容量一般在 4～128 字节之间，命中时间可以达到 1 个时钟周期的时间，失效时间取决于主存的访问时间及传输时间，一般为 8～32 个时钟周期，失效率在 1%～20%之间。为了提高 Cache 数据的调入、调出速度，要求主存的带宽与其匹配。

3.4. 虚拟存储器

虚拟存储器是存储器组织中的一个重要概念。在以上论述中都假定 CPU 用地址信号直接指定存储器中的实际存储位置，这样在计算机中运行多个不同的程序时就要求各个程序使用不同的地址范围。采用虚拟存储器后，实际上计算机在访问存储器时都不是直接根据程序员指定的地址进行的。存储系统能够将程序员指定的地址转换成可在存储器中访问的地址。程序指令生成的地址是虚拟地址，又称为逻辑地址；经过转换后的地址是实际地址，又称物理地址。虚拟地址的范围称为虚拟地址空间，它是程序员所看到的地址空间。程序员使用的虚拟地址空间经过转换后，形成存储器的物理地址空间。计算机中这种虚拟地址与实际地址的映射关系可以在运行过程中根据系统的要求动态改变。通过将不同用户程序的逻辑地址空间转换成不同的物理地址空间，系统可将用户程序的存储空间相互隔离，从而保护存储空间。存储空间的保护是虚拟存储器的重要用途之一，它使得每个用户进程之间可以实现存储空间的相互隔离和有限制的共享。虚拟存储器可以通过地址转换时检查构成的物理地址的范围和访问方式（读、写或执行等），以及访问的主体的识别，对访存操作进行保护。

虚拟存储器另一个重要用途是解决计算机中主存储器的容量问题，要求在不明显降低平均访存速度的前提下增加程序的访存空间。在计算机中经常会遇到主存储器容量不够的问题。增加存储器容量的一个经济有效的方法是使用磁盘等外存来构成运行中所需要的程序和数据的存储空间，使得虚拟地址能够映像到磁盘的存储空间。在将磁盘的存储区域移到内存中后，就使得程序能够像访问主存储器一样访问外部存储器。而在没有虚拟存储器的计算机中，程序和数据必须全部装入内存后才能运行，程序只能使用内存中的指令和数据。虚拟存储器使得程序的地址空间最大可以达到 CPU 的虚拟地址范围，即 CPU 的最大存储空间取决于虚拟地址位数。为了不明显降低访存速度，将虚拟地址空间中的访问最频繁的一小部分地址范围映射到主存储器，其余的地址空间映射到外存储器。这样，从程序员的角度看，存储空间扩大了。虚拟存储器使存储系统具有外存的容量又有接近于主存的访问速度。当程序要访问的存储单元映射到主存中的空间时，存储系统将虚拟地址转换成主存的实际地址，如果虚地址对应的存储单元不在主存中，则不能直接形成物理地址。这时存储系统先将要访问的空间映射到主存，并将该存储单元从外存调入主存，然后改变虚拟地址与实际地址的映射关系，进行地址转换后对数据进行访问。

虚拟存储器还简化了程序的装入运行，它提供了程序的重定位能力。因为程序使用的虚拟地址可映射到不同的物理地址。这就使得程序能够装入到主存中的任意位置。

虚拟存储器主要是由软件进行管理的，而 Cache 是由硬件管理的。在计算机中，各种存储器硬件以及管理这些存储器的软硬件构成了计算机的存储系统。存储器管理软件属于操作系统的一部分。硬件对于虚拟存储器的支持主要是提供快速的地址转换，这部分硬件称为存储器管理单元（MMU)。虚拟存储器与 Cache 存储器的管理方法有许多类似之处，但由于历史的原因，它们各自使用不同的术语。虚拟存储器中的数据管理的基本单位称为“页”或“段”，如果页不存在于主存中，则称为“页失效”。虚拟存储器和 Cache 两者间主要区别在于以下几方面：

（1）Cache 的替换策略是由硬件实现的，而虚拟存储器的替换策略主要是由操作系统实现的。因为虚拟存储器中的替换操作涉及外围设备的操作，需要花费很长时间，允许系统用较长的时间确定替换哪一个页。操作系统可用较多的时间采用较好的算法确定替换的页，而操作系统在替换策略中所花费的时间相对于一次磁盘的访问时间很少。

（2）虚拟存储器中一般使用全相联地址映射方式以提高命中率。由于主存与磁盘等外存的工作速度差异很大，所以对虚拟存储器来说，提高主存的命中率特别重要。

（3）虚拟存储器中一般采用写回法更新策略，并且是按写分配的。程序的写操作只对内存进行，不直接写磁盘上的页。

（4）CPU 的存在及其所有的操作对程序员一般都是透明的，虚拟存储器中页在主存与外存之间的传输对系统程序员是不透明的，而对应用程序员和用户是透明的，段对应用程序员可透明也可不透明。

（5）虚拟存储器的存储空间大小受到计算机地址空间的限制，即虚拟地址的位数， 而 Cache 存储器的容量以及主存的容量则一般远小于的地址空间，因而不存在这种限制。

在虚拟存储器每次访问存储器时，都需要将虚地址转换成主存中的实际地址，如果对应的存储页（虚拟页或逻辑页）不在主存中，则需要将其调入主存。虚地址到的映像方法以及替换算法与 Cache 中的类似，用建立和查找页表的方式进行。每个程序有它自己的虚拟地址空间，使得每个进程的地址空间相互隔离。 根据采用的存储器地址映像算法，可将虚拟存储器的管理方式分成段式、页式和段页式等多种。这些管理方式的基本原理是类似的。

习题 3

1.解释下列术语。

EDO DRAM　　PROM　　EPROM　　EEPROM　　SDROM

快闪存储器　　相联存储器字地址　　字节地址　　块地址

2.EPROM 是不是 PLD？它是否可以用于实现任意组合逻辑？是否可以用于实现任意时序逻辑？假定有足够的输人信号端和输出信号端。

3．电可擦写的 ROM 中存储的信息可以任意擦除并修改，它是否可以代替 RAM 成为计算机的主存用芯片？

4.ROM 存储器中的存储单元能否被随机地访问？

5.存储器芯片的容量通常用 aXb 的方式表示，其中 a 为字数，b 为每个字的位数。以下几种存储器芯片分别有多少地址线和数据线？

（1）2K×16。

（2）64K×8。

（3）16M×32。

（4）4G×4。

6．用 4K×8 的存储器芯片构成 16×8 的存储器，地址线为 A15～A0，请写出全部片选信号的逻辑表达式。

7．用 4K×8 的存储器芯片构成 8K×16 位的存储器，共需多少片？如果 CPU 的信号线有读写控制信号 R/W*，地址线为 A15～A0，存储器芯片的控制信号有 CS*和 WE*，请画出此存储器与 CPU 的连接图。

8.用 64×1 位的芯片设计一个总容量为 1024 字节的 16 位存储器，画出逻辑图并指出所需的所有输入和输出信号。

9.假定计算机系统需要 512 字节 RAM 和 512 字节 ROM 容量。使用的 RAM 芯片是 128 字×8 位，ROM 芯片为 512 字×8 位。RAM 芯片有 CS*及 WE*控制端，ROM 芯片有控制端，有地址线 A15～A0、数据线 D7～D0、读写控制线 RW*等，试确定各存储器芯片的地址区间，指出存储器以及各存储器芯片需要的地址线数量，并画出存储器与 CPU 的连接图。

10.某 32 位计算机系统的主存采用 32 位字节地址空间和 32 位数据线访问存储器，若使用 4M 位的 DRAM 芯片组成 32MB 主存，并采用内存条的形式，问：

（1）若每个内存条为 4M×32 位，共需多少内存条？

（2）每个内存条内共有多少片 DRAM 芯片？

（3）主存共需多少芯片？

（4）如何有选择地访问各内存条？

11.设有若干片 1K×8 位的 SRAM 芯片，问如何构成 2K×32 位的存储器？画出该存储器与

CPU 连接的结构图，设 CPU 的接口信号有地址信号、数据信号、控制信号 MREQ#和 R/W#。

12.某计算机 4K×8 位的主存地址空间中用 2 片 1K×8 的 ROM 和 2 片 2K×4 的芯片构成。画出 CPU 与 RAM 和 ROM 连接图。RAM 的控制信号为 CS#和 WE#，CPU 的地址线为 A11～A0、，数据线为 8 位的线路 D7～D0。，控制信号有读写控制 R/W#和访存请求 MREQ#。

13.用 64×1 位的 SRAM 芯片设计一个总容量为 1024 字节的 16 位存储器，画出逻辑图并指出所需的所有输入和输出信号，要求该存储器既能以字节方式访问，又能以 16 位的字方式访问。

14.某计算机的主存地址空间中，从地址 000016～3FFF16 为 ROM 存储区域，从 400016～5FFF16 为保留地址区域，暂时不用，从 600016～FFFF16 为 RAM 地址区域。RAM 的控制信号为 CS#和 WE#，CPU 的地址线为 A15～A0，数据线为 8 位的线路 D7～D0，控制信号有读写控制 R/W#和访存请求 MREQ#，画出地址译码方案，如果 RAM 和 ROM 存储器芯片都采用 8K×1 的芯片，试画出存储器与的连接图。

15. 一台计算机采用 256×8 的 RAM 芯片和 1024×8 的 ROM 芯片。计算机系统需要 2K 字节的 RAM 和 4K 字节的 ROM，以及 4 个输入/输出接口，每个接口有 4 个 8 位的寄存器，采用存储器映像的编址方式，位于 8KB 地址空间的高端。存储器地址的最高 2 位为 00 表示访问 RAM，为 01 或 10 表示访问 ROM，为 11 表示访问输入/输出接口寄存器。 、

(1)需要多少 RAM 和 ROM 芯片？

(2)画出存储器地址映像表，指出地址空间中各段分别映像到什么芯片。

(3)用十六进制数给出 RAM、ROM 和接口寄存器的地址范围。

16.Intel 82875 MCH 存储器控制接口支持 128Mb、256Mb、512Mb 的 8 位或者 16 位的芯片，存储器数据接口为 64 位，问：

(1)存储器容量最小是多少？

(2)将存储器芯片以位扩展方式构成内存条，再字扩展方式扩展容量，最多支持 8 个内存条，存储器容量最大是多少？

(3)最大配置时，每次存储器刷新多少数据位？

17.Cache 的命中率与哪些因素有关？它们如何影响 Cache 的命中率？

第4章. 输入输出系统

4.1. 概述

计算机的输入输出系统主要包括输入输出设备，输入输出接口、输入输出方式和计算机总线。如图 4-1 所示，现代计算机系统常采用总线结构， I/O 设备通过接口模块挂在系统总线上，通过 I/O 总线与主机相连。

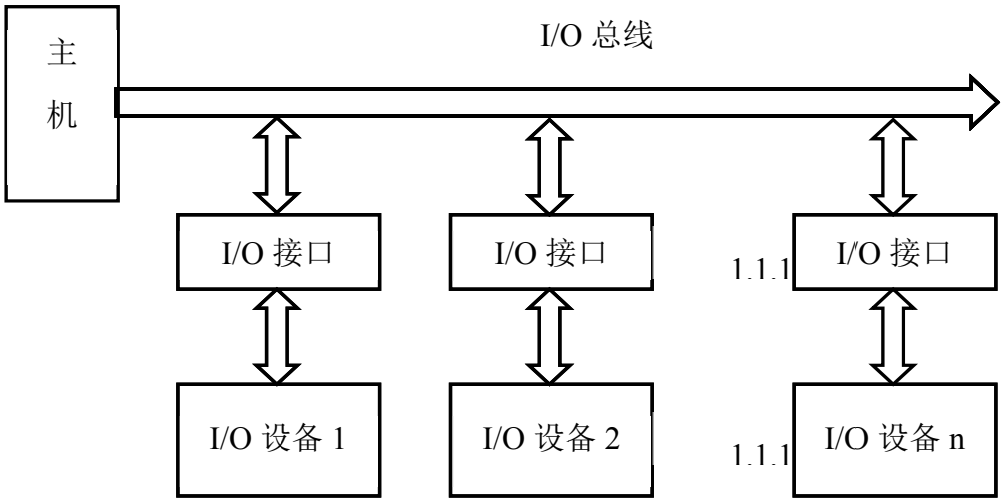


图 4.1 外部设备通过接口和 I/O 总线与主机交换信息

输入输出设备，也称作计算机的外部设备，指的是能完成特定功能、同时又相对独立的精密机械电子设备，其接口设计面向计算机的外部接口，可完成输入或输出功能。如键盘、显示器、打印机等。

输入输出接口，又称 I/O 接口，则是位于计算机总线和输入输出设备之间，用于在计算机主机和输入输出设备之间实现信息交换的电子电路。该部分电路常常基于某种可编程的接口芯片来设计，可以完成诸如数据格式变化、电平转换、速度匹配、命令和反馈信号交互等功能。

输入输出方式，是指输入输出设备以何种方式和计算机主机进行数据交换。常用的输入输出方式有程序控制方式（查询方式、中断方式）、直接内存访问（DMA）方式和通道。

I/O 总线是计算机主机与外部设备交换信息的公共通道，它将计算机主机和输入输出设备在物理上连接为一体，构成一个能完成信息输入和数据输出的完整的计算机硬件系统。

4.2. 输入输出设备

4.2.1. 输入输出设备分类

输入输出设备可分为三类：

一．人机交互设备

输入设备：键盘、图形图像输入、条形码、语音文字输入设备等。

输出设备：打印机、显示器、绘图仪、语音合成器等。

二．计算机信息的驻留设备

磁盘、磁带、光盘、穿孔纸带等。

三．机－机通信设备

MODEM、网卡、集线器、网络交换机、路由器等。

本节主要介绍人机交互设备，主要分为输入设备和输出设备两类，部分设备同时具备输入和输出功能。

4.2.2. 输入设备

计算机的输入设备很多，如有键盘、鼠标、触摸屏、扫描仪等。下面简要介绍其中的几种的基本原理。

一、键盘

键盘是最常用的输入设备。通过键盘上的各个键，可以向主机输入各种信息，如汉字、字母、数字及标点符号等。

键盘是由一组排列成阵列形式的按键开关组成。如图 4.2（a）所示的 4×4 行列式键盘。每一次通过键盘向计算机输入信息都包含下列三个步骤：

按一下键；

查出按的是哪个键；

将此键翻译成 ASCII 码，发送给计算机。

要实现上述功能，还需要键盘接口（接口电路或接口软件），键盘接口需完成去抖动，防串键，按键识别与键码产生等功能：

1. 去抖动：如图 4.2（b）所示，按键在被按下或释放后的一段时间内（t0 和 t1 通常在毫秒级）会出现机械的抖动，识别按键按下或释放动作时必须避开抖动状态，只有按键处在稳定接通或断开状态时，才能保证识别正确无误。

2．防串键：防串键是为了解决多个键同时按下或者前一键没有释放而又有新键按下时产生的问题。

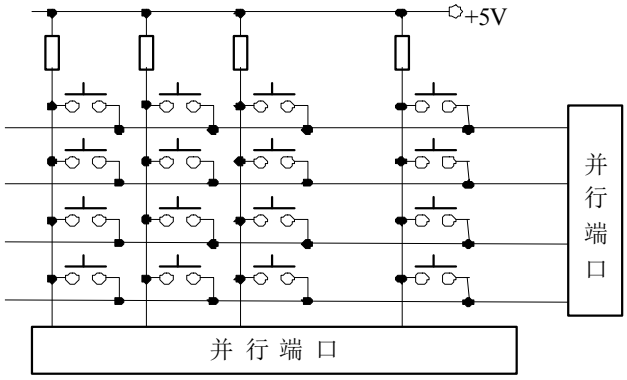
3．按键识别：指如何识别被按键，常用行描法和线反转法。

4．键码产生：为了从键的行列坐标编码得到反映键功能的键码，一般在内存中定义一个键位编码表，通过查表获得键码。

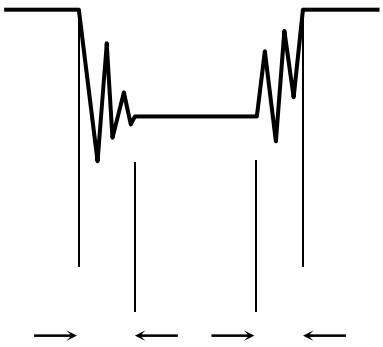
确认是哪一个键按下，可用硬件方法，也可用软件的方法。

采用硬件确认是哪一个键按下的方法称为编码键盘法，采用软件确认是哪一个键按下的方法称为非编码键盘法。

编码键盘与非编码键盘的主要区别是：编码键盘本身带有实现接口主要功能所需的硬件电路，不仅能自动检测被按下的键并完成去抖动防串键等功能，而且能提供与被按键功能对应的键码（ASCII 码）送往 CPU，而非编码键盘只简单的提供按键开关在行列矩阵中的位置，有关键的识别、键码的确定以及去抖动等功能场由软件完成。



（a）4×4 行列式键盘



（b）按键和释放按键的抖动

二．鼠标

鼠标(Mouse)是一种手持式的定位设备。图 4.3 是机械式鼠标的原理图。

机械式鼠标的的底座装有一个胶球，球在桌上摩擦使球转动，胶球滚动带动两个滚轮（水平轮和垂直轮）。水平轮探测 X 方向的运动，垂直轮用来探测 Y 方向的运动。当球滚动时，这两个滚轮的一个或者两个跟着旋转。每个滚轮连着一根轴，这根轴上固定了一个有孔的圆盘。当滚轮滚动时，轴和圆盘一块跟着旋转。在圆盘的两侧有一个红外发光二极管和一个红外传感器。圆盘转时外围的间隔孔阻隔了发光二级管发出的光线，因此另一边的传感器就接受到了光脉冲。脉冲的频率直接和鼠标移动的距离和速度相关。处理芯片读取红外传感器的脉冲并且把它转换成二进制，然后通过鼠标电缆将此二进制数据送给电脑。

另一种是光电式鼠标，它需要与一块画满小方格的长方形金属板配合使用。安装在鼠标底部的光电转换器可以确定坐标点的位置。光电式鼠标比机械式鼠标可靠性高，但需要增加一块金属板。

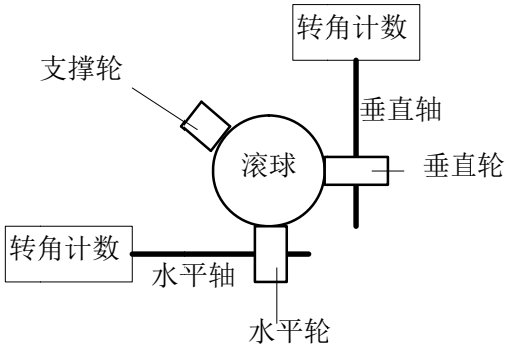


图 4.3 机械式鼠标原理图

三．触摸屏

触摸屏主要分电阻、电容式、红外式和感应式触摸屏。

电阻式触摸屏结构如图 4.4 所示，它主要是由两层相互绝缘的透明阻性薄膜（塑料膜上喷涂 ITO）组成的，两导电薄膜中间有许多透明的绝缘隔离点。内层阻性薄膜作偏置层（上边沿和左边沿接参考电压 Vref，下边沿和右边沿接参考地 GND）。当外面一层软膜被按下并与内层导电膜接触后，X 方向在接触点处被分隔成左右两个分压电阻，Y 方向的电阻薄膜则在接触点处被分隔成上下两个分压电阻，接触点的输出电压 Vx 和 Vy 被送到一个高输入阻抗的 ADC 的 X+和 Y+，A/D 转化的结果送 CPU 处理，就可以确定接触点的坐标。

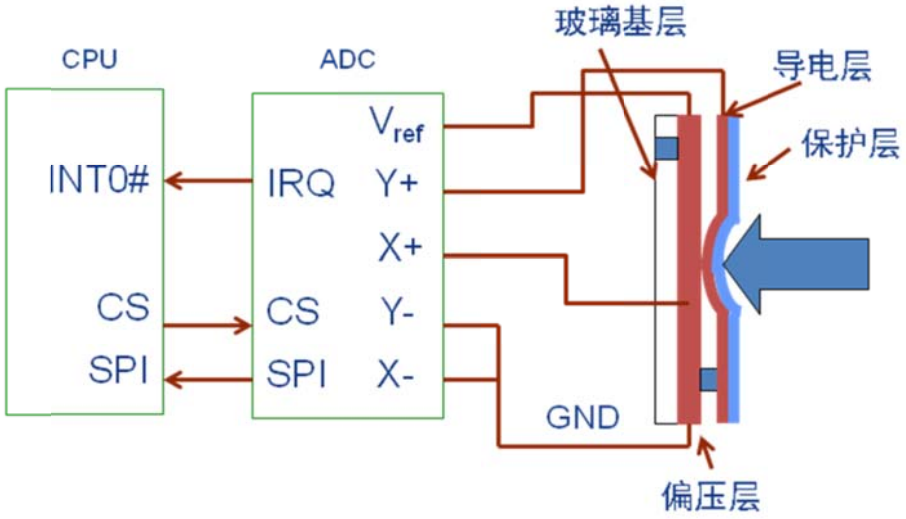


图 4.4 电阻式触摸屏原理图

红外触摸屏在显示器的前面安装一个电路板外框，如图 4.5，电路板外框四边均匀布置了红外线发射管和红外接收管，发射管和接收管一一对应构成横竖交叉的红外矩阵。用户在触摸屏幕时，手指就会挡住经过该位置的横竖两条红外线，根据 X 方向和 Y 方向红外接收管的输出就可以判断出触摸点在屏幕的位置。

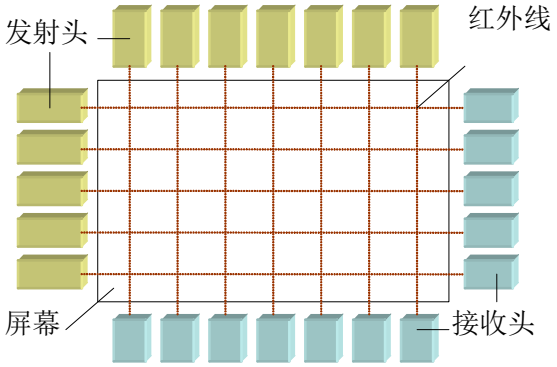


图 4.5 红外触摸屏结构原理

4.2.3. 输出设备

计算机的输出设备也很多，这里介绍最常用的两种，显示器和打印机。

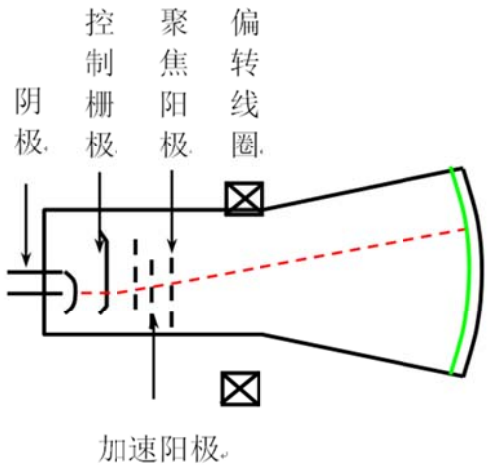


图 4.6　CRT 的结构原理

1．CRT 组成及工作原理

CRT 是目前应用最广泛的显示器件，既可作为字符显示器，又可作为图像、图形显示器。CRT 是一个漏斗形的电真空器件，由电子枪、荧光屏及偏转线圈组成，如图 4.6 所示。

电子枪包括灯丝、阴极、控制（栅）极、加速阳极、聚焦阳极。当灯丝加热后，阴极受热而发射电子，电子的发射量和发射速度受控制极控制。电子经加速、聚焦而形成电子束，在第三阳极形成的均匀空间电位作用下，使电子束高速射到荧光屏上，荧光屏上的荧光粉受电子束的轰击产生亮点，其亮度取决于电子束的轰击速度、电子束电流强度和荧光粉的发光效率。电子束在偏转系统控制下，可在荧光屏的不同位置产生光点，由这些光点可以组成各种所需的字符、图形和图像。

彩色 CRT 的原理与单色 CRT 的原理是相似的，只是对彩色 CRT 而言，通常用 3 个电子枪发射的电子束，经定色机构，分别触发红、绿、蓝三种颜色的荧光粉发光，按三基色迭加原理形成彩色图像。

分辨率和灰度等级是 CRT 的两个重要技术指标。分辨率是指显示屏面能表示的像素点数分辨率越高，图像越清晰。灰度等级是指显示像素点相对亮暗的级差，在彩色显示器中它还表现为色彩的差别。

CRT 荧光屏发光是由电子束轰击荧光粉产生的，其发光亮度一般只能维持几十毫秒。为了使人眼能看到稳定的图像，电子束必须在图像变化前不断地进行整个屏幕的重复扫描，这个过程称为刷新。每秒刷新的次数称为刷新频率，一般刷新频率大于 30 次 / 秒时，人眼就不会感到嘎烁。在显示设备中，通常都采用电视标准，每秒刷新 50 帧(Frame)图像。

为了不断地刷新，必须把瞬时图像保存在存储器中，这种存储器称为刷新存储器，又称帧存储器或视频存储器(Video RAM)。刷新存储器的容量由图像分辨率和灰度等级决定。分辨率越高，灰度等级越多，需要的刷新存储器容量就越大。例如，分辨率为 512×512 像素，灰度等级为 256 的图像，其刷新存储器的容量需达 512×512x8 b，即为 256 KB。

计算机的显示器大多采用光栅扫描方式。所谓光栅扫描，是指电子束在荧光屏上按某种轨迹运动，光栅扫描是从上至下顺序扫描，可分为逐行扫描和隔行扫描两种。一般 CRT 都采用与电视相同的隔行扫描，即把一帧图像分为奇数场（由 1、3、5 等奇数行组成）和偶数场（由 0、2、4、6 等偶数行组成），一帧图像需扫描 625 行，则奇数场和偶数场各扫描 312.5 行。扫描顺序是先扫描偶数场，再扫描奇数场，交替进行，每秒显示 50 场。

2．字符显示器

字符显示器是计算机系统中最基本的输出设备，它通常由 CRT 控制器和显示器(CRT)组成，图 4.7 示意了它的原理框图。

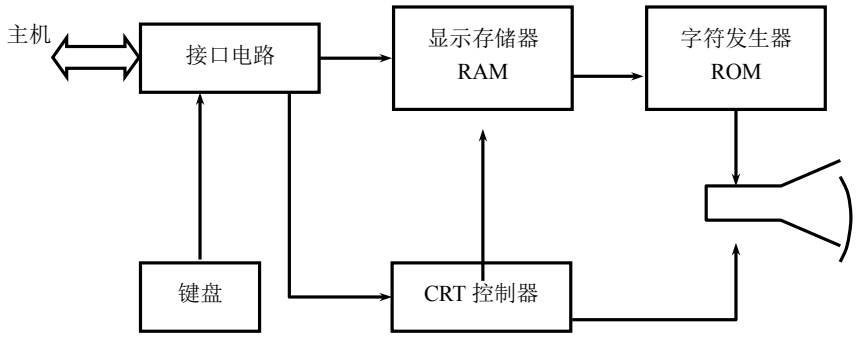


图 4.7 字符显示器原理框图

1）显示存储器（刷新存储器）

显示存储器用于存放欲显示的字符，其容量与一屏显示的字符数有关。常用的 80 列×25 行字符显示器，显示存储器容量至少 2000 字符（字节）。由于字符编码一般采用 7 位的 ASCII 码，所以未用的每个字节的最高位（D7）可以用于指示字符的闪烁特性。

显示存储器的地址与屏幕字符显示的坐标一致，对于位于坐标（Row，Col）处的字符，其在 VRAM 中的地址为 80*Row+Col，其中，坐标原点（0，0）在左上角。

2）字符发生器

字符发生器用于产生显示字符用的点阵数据。

由于荧光屏上的字符由光点组成，而显示存储器中存放的是 ASCII 码，因此，必须有一个部件能将每个 ASCII 字符码转变为一组 5x7 或 7x9 的光点矩阵信息。具有这种变换功能的部件称为字符发生器，它实质是一个 ROM。图 4.8 是一个对应 7x9 光点矩阵的字符发生器原理框图。

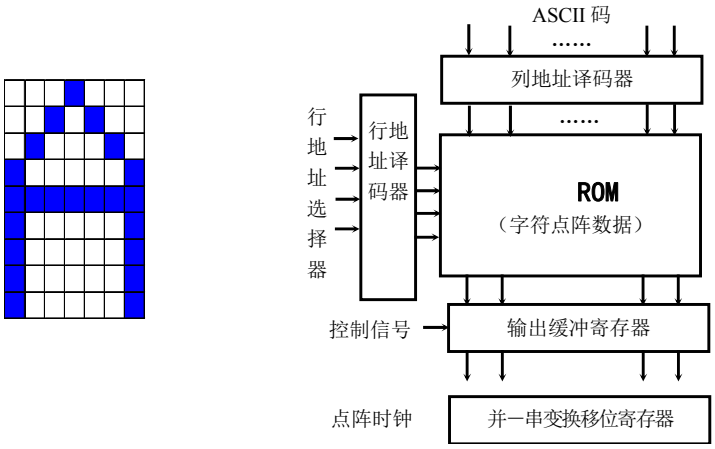


图 4.8 字符点阵字符发生器原理图

3） CRT 控制器

功能：

- 接收 CPU 的数据和控制信号
- 产生字符 RAM 地址和字符 ROM 的地址信号
- 产生 CRT 的水平同步信号和垂直同步信号

二．打印机

打印设备可以分为下列两种类型：

1） 击打式：活字式/点阵针式；工作方式：串行/并行

2）非击打式：电、磁、光、喷墨、化学方法（热敏打印）等

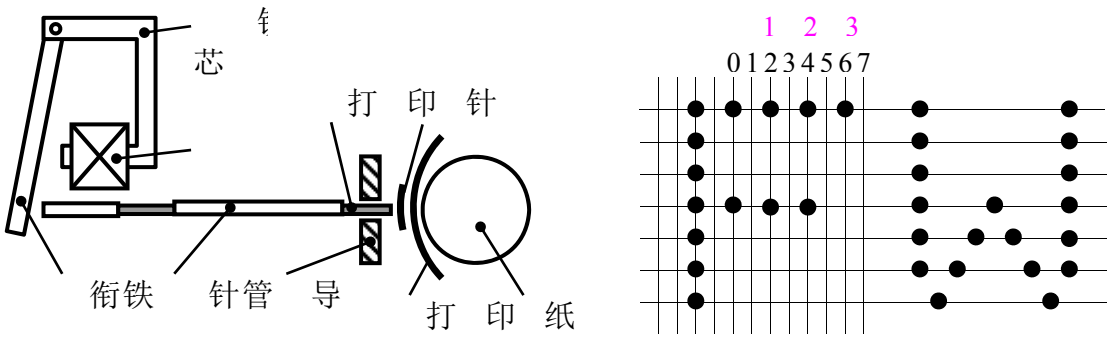
1．点阵针式打印机

点阵针式打印机结构简单、体积小、重量轻、价格低、字符种类不受限制、较易实现汉字打印，还可打印图形和图像，是目前应用最广泛的一种打印设备。一般在微型、小型计算机中都配有这类打印机。

点阵针式打印机的印字原理是由打印针（钢针）印出 nxm 个点阵来组成字符或图形。点多、越密，字形质量越高。西文字符点阵通常采用 5x7、7x7、7x9、9x9 几种，汉字的点阵采 16x16、24×24、32x32 和 48x48 多种。图 4.9 是 7x9 点阵字符的打印格式和打印头的示意图

打印头中的钢针数与打印机型号有关，有 7 针、9 针，也有双列 14(2 x7)针或双列 24(2 12)针。打印头固定在托架上，托架可横向移动。图 4.9 中为 7 根钢针，对应垂直方向的 7 一由于受机械安装的限制，这 7 点之间有一定的间隙。水平方向各点的距离取决于打印头移动位置，故可密集些，这对形成斜形或弧形笔画非常有利。字符的形成是按字符中各列所包含的逐列形成的。例如，对于字符 E，先打印第 2 列的 l—7 个点，再打印第 4、6、8 列的第 1、4、7 点，最后打第 10 列的 1、7 两个点。可见每根针可以单独驱动。打印一个字符后，空

出 3 列(11、0、1 列)作为间隙。



（a）打印头示意图　　（b）点阵字符的打印格式

图 4.9 针式打印头和打印格式示意图

针式打印机由打印头、横移机构、输纸机构、色带机构和相应的控制电路组成，如图 4.10 所示。

打印机被 CPU 启动后，在接收代码时序器控制下，功能码判别电路开始接收从主机送来的欲打印字符的字符代码（ASCII 码）。首先判断该字符是打印字符码还是控制功能码（如回车、换行、换页等），若是打印字符码，则送至缓冲存储器，直到把缓冲存储器装满为止；若是控制功能码，则打印控制器停止接收代码并转入打印状态。打印时首先启动打印时序器，并在它控制下，从缓冲存储器中逐个读出打印字符码，再以该字符码作为字符发生器 ROM 的地址码，从中选出对应的字符点阵信息（字符发生器可将 ASCII 码转换成打印字符的点阵信息）。然后在列同步脉冲计数器控制下，将一列列读出的字符点阵信息送至打印驱动电路，驱动电磁铁带动相应的钢针进行打印。每打印一列，固定钢针的托架就要横移一列距离，直到打印完最后一列，形成 n×m 点阵字符。当一行字符打印结束或换行打印或缓存内容已全部打印完毕时，托架就返回到起始位置，并向主机报告，请求打印新的数据。

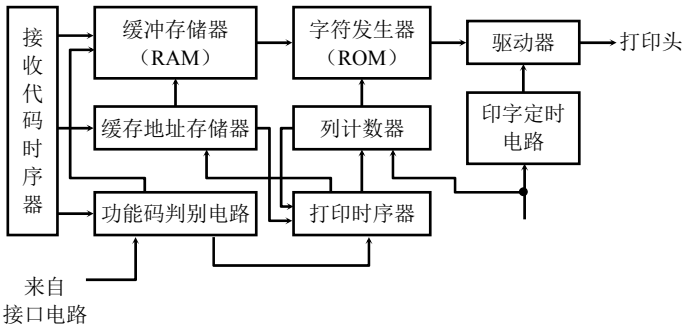


图 4.10 针式打印机控制电路框图

2．激光打印机

1）．激光打印机的组成

激光打印机是激光技术和电子照相技术相结合的产物。它由走纸机构、激光扫描系统、电子照相部分和印字字形发生器与控制器等几部分组成。如图 4－11 所示。

激光扫描系统的功能，是控制激光束能扫描到字形鼓柱面的任何位置，它由激光器，偏转调制器，扫描器和光路系统组成。激光器大部分使用氦—氖气体激光器，它提供打印机运行所使用的光源。偏转调制器通常用声光器件（在器件内，用超声波改变媒质对光的衍射特性来改变光线的传播方向）调制激光束的传播方向，扫描器实现激光束沿字形鼓的轴线重复做横向移动，激光束的纵向移动是靠字形鼓的旋转完成的。这样，通过字形鼓的旋转和激光束的水平移动，就可以扫描到字形鼓柱面的任何位置。

电子照相部分的核心部件是圆柱型的字形鼓，又称光导鼓，柱面高度光洁，镀有一层由硒—碲合金组成的具有良好光导特性（光线照射后电阻率降为原来的 1/100 到 1/1000）的材料，用于完成对打印内容的照相、显影和转印。

2）激光打印机的印字过程

准备阶段，开始时，通过电晕放电装置（用于使附近的空气电离）对光导鼓表面均匀的充上一层带正电的空气离子，其表面电位可达几百伏，光导材料的内层感应出负电荷的电子，在没有光线照射的条件下，二者隔着光导层互相吸引，既不回中和也不会离去。

照相阶段，在由被打印信息控制而提供出来的激光束扫描光导鼓时，光导鼓的不同部位就会发生不同的变化，在激光束照射到的明区电阻率降低，该处的电荷将会放掉，激光束未照射到的暗区的带电情况不变，这些剩下来的静电区域就是被打印信息的潜像。

显影阶段，用的是墨粉和表面涂有树脂薄膜、直径为几百微米的玻璃珠为载体的混合物，运动中他们互相摩擦产生静电，墨粉被吸附在载体表面，当这些载体流过正经过这里的带有静电潜象信息的光导体表面时，载体表面上的墨粉被潜像的静电电荷所吸引，离开载体而留到了光导体表面，从而形成了由墨粉显示出来的字形。

转印阶段，完成把光导体表面的字形墨粉转移到打印纸上，多数采用在打印纸的背面（打印纸的另一面靠贴到光导体表面）通过电晕放出与墨粉所带电荷极性相反、电位更高的电荷，通过强力磁场把墨粉抢到打印纸上来，这就在打印纸上有了静电吸附着的墨粉字形。

定影阶段，是把墨粉牢靠永久固定在打印纸上的工作。这是通过红外光加热或辐射加热的办法，用 100 摄式度的温度把墨粉熔化并凝沾在打印纸上，从而完成完整的打印过程。

激光打印机属于页式打印机，光导鼓每旋转一周打印一页内容。在开始下一页打印前，还要由清扫器清除光导鼓表面上剩余的墨粉，消电灯消除光导鼓上残存的电荷。激光打印机打印速度快（每分钟几页、十几页，几十页甚至更多），印字质量高，噪音低，有普及型和各种高档型产品，被广泛应用在许多场所。

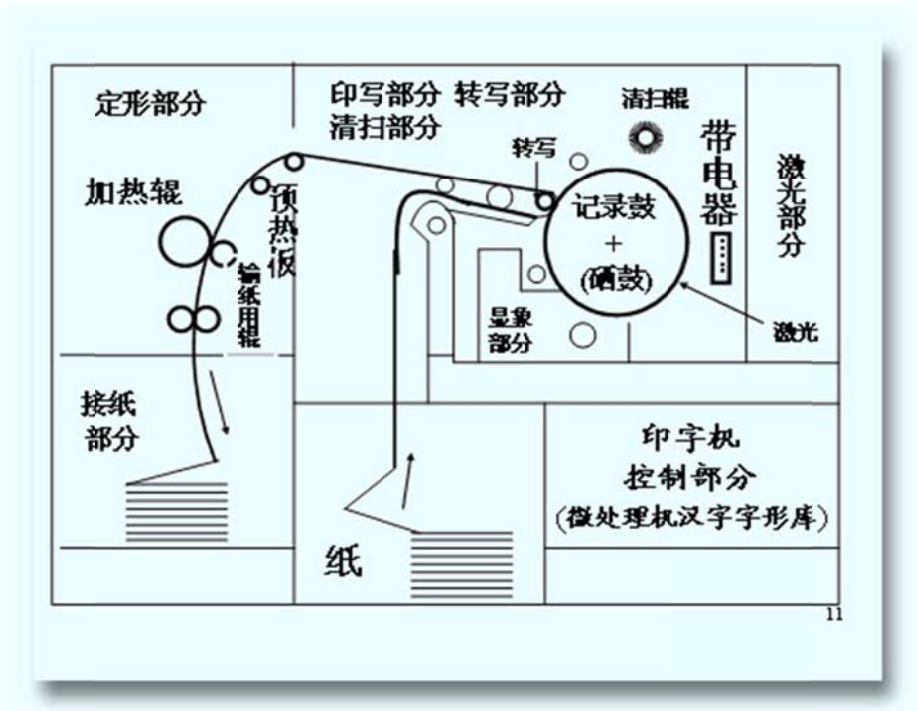


图 4.11 激光打印机原理框图

4.3. 输入输出方式

在计算机主机和 I/O 设备之间，可以采用不同的控制方式进行数据传送。微型计算机常与 I/O 设备的输入输出方式又查询方式（Polling）、中断（interrupt）方式、直接存储器存取方式（DMA），在大型计算机中还会采用 I/O 通道控制方式和 I/O 处理机，它们在性能、价格、适用场合等各方面都不一样。

4.3.1. 查询方式

1．查询方式及工作流程

查询方式是指 CPU 在某程序循环体中通过 I/O 输入指令不断查询设备的运行状态（Busy or idle），来控制数据输入或输出的一种方式。采用这种方式实现主机和 I/O 设备交换信息，要求 I/O 接口内设置一个能反映 I/O 设备是否准备就绪的状态信号，CPU 通过对状态信号的检测来判断 I/O 设备的准备情况。

举例：带 Centronics 并行接口的打印机都有一根 BUSY 状态信号，当 BUSY 有效（低电平）时，表示打印机正忙，还不能接收新的打印字符。当 BUSY 信号无效时，计算机可以向打印机传送新的打印字符。图 4.12 是采用查询方式打印多个字符的程序流程图。由这个查询过程可见，CPU 在循环体内反复查询“BUSY”状态信号，这个阶段，CPU 不能执行其它程序，可见这种方式使 CPU 和打印机处于串行工作状态，CPU 的工作效率不高。

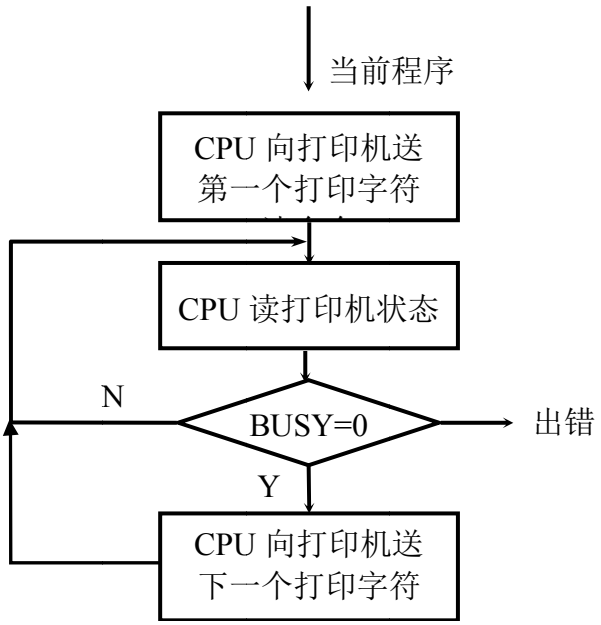


图 4.12 程序查询方式流程图

4.3.2. 中断方式

1． 中断的概念

计算机在程序执行过程中，当出现某种随机事件（如掉电或 I/O 设备准备就绪）并希望

CPU 作出立即响应的时候，可以通过一根特殊的信号线（中断请求 INTR）向 CPU 发出请求，计算机停止当前程序的运行响应这个中断请求，转向执行一段被称为中断服务子程序（ISR）的程序段，ISR 运行结束后再返回原断点处继续，这就是“中断”的一般工作方式。实现中断功能所需要的软硬件技术，统称为中断技术。

与查询方式相比，中断技术可以显著提高 CPU 的工作效率，同时也加快了对外部中断请求的响应速度。图 4.14 是表示 CPU 在运行程序的同时通过中断方式完成打印任务的示意图。对于某些不可预测的紧急事件，如掉电、设备故障保护等，要求 CPU 迅速做出处理，也常常使用中断技术。

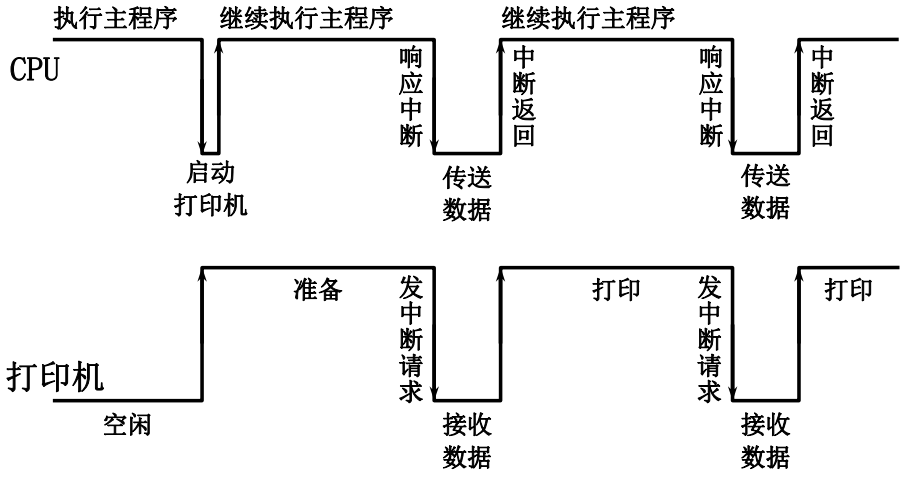


图 4.13 CPU 与打印机并行工作时间示意图

从图 4.13 可以看出，采用中断方式传送信息时，CPU 在启动 I/O 设备后，不去查询 I/O 设备的状态，而是继续运行主程序。只有当 I/O 设备准备就绪后才向 CPU 提出申请服务的请求（中断请求），CPU 在响应中断请求后（中断响应），停止当前运行的程序，然后进入中断服务程序，在中断服务程序中 CPU 与 I/O 设备交换信息，交换完毕后返回被中断的程序继续。中断方式下，CPU 与 I/O 设备绝大多数时间都并行工作，交换信息通常只需要数条指令，所用时间开销比查询方式大大缩短。

程序中断方式需要解决中断现场保护和中断现场恢复问题，才能保证程序能正确返回并得到正确的执行。堆栈的数据结构非常适合现场保护及现场恢复。

2．中断的响应过程

当 CPU 响应中断的条件满足后，CPU 就会响应中断，转入中断程序处理。具体的工作过程如下所述。

1) 关中断：CPU 响应中断后，发出中断响应信号的同时，内部自动地实现关中断。

2）保留断点：CPU 响应中断后，把主程序执行的位置和有关数据信息保留到堆栈，以备中断处理完毕后，能返回主程序并正确执行。

3）保护现场：为了使中断处理程序不影响主程序的运作，故要把断点处的有关寄存器的内容和标志寄存器推入堆栈保护起来，即在中断服务程序中把这些寄存器的内容推入堆栈。这样，当中断处理完成后返回主程序时，CPU 能够恢复主程序的中断前状态，保证主程序

的正确动作。

4）给出中断入口，转入相应的中断服务程序：系统由中断源提供的中断向量形成中断入口地址，使 CPU 能够正确进入中断服务程序。

5）恢复现场：把所保存的各个内部寄存器的内容和标志位的状态，从堆栈弹出，送回 CPU 中原来的位置。这个操作在系统中也是由服务程序来完成的。

6）开中断与返回：在中断服务程序的最后，要开中断（以便 CPU 能响应新的中断请求）和安排一条中断返回指令，将堆栈内保存的主程序被中断的位置值弹出，运行被恢复到主程序。

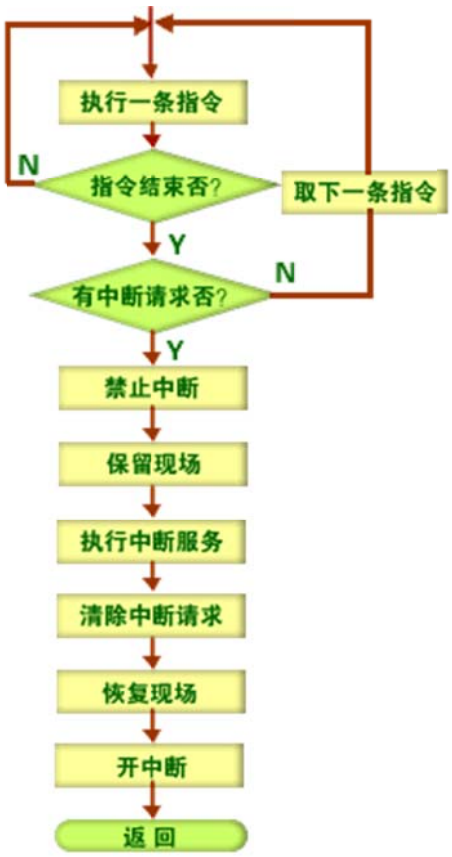


图 4-14 中断响应流程图

4.3.3. DMA 方式

DMA（Direct Memory Access－直接存储器存取）提供了一条 I/O 设备与主存直接交换数据的通道。若 I/O 接口的 DMA 控制器提出请求时，CPU 将总线权让给 DMA 控制器，DMA 控制器通过总线直接控制 I/O 设备与主存交换信息。DMA 数据传送过程是在 DMA 控制器的控制下来完成的，不需要 CPU 的介入。

通常把 DMA 这种对总线的占用称为“窃取”或“挪用”，“窃取周期”/“挪用周期”一般为一个存储周期。

在 DMA 挪用周期内，CPU 仍可进行内部操作，如算术运算等。

1. DMA 方式特点

与查询方式相比，程序中断方式显著地提高了 CPU 的工作效率。但从微观操作分析，CPU 在处理中断服务程序时，需要暂停当前运行的程序，同时对程序断点进行保护，以便中断服务程序运行结束后能准确返回，因此还是需要增加一些 CPU 的额外开销。对于高速 I/O 与存储器之间的大批量数据交换，若采用程序中断方式,将不断打断 CPU 的正常运行，使 CPU 工作效率难以进一步提高。

实际上，在 I/O 设备与存储器之间的数据交换过程中，CPU 可以不必介入。若 I/O 端口能直接访问存储器，可以进一步提高 CPU 的工作效率，由此产生 DMA 方式。

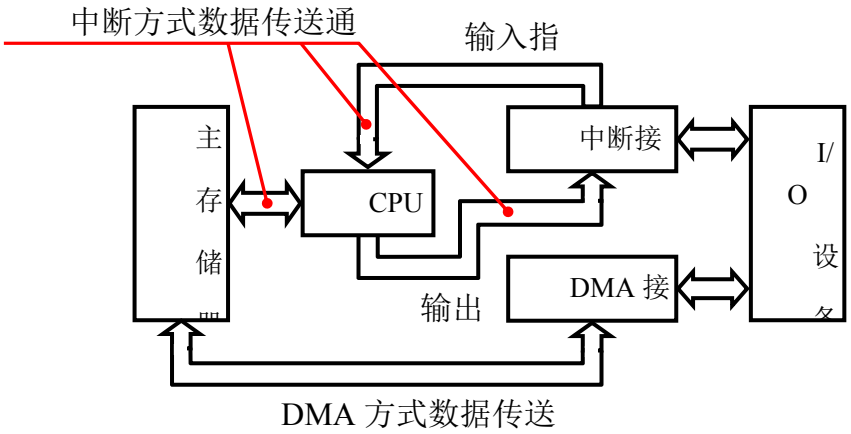


图 4.15 DMA 方式和程序中断方式的数据通道

由图 4.15 可知，主存与 DMA 接口之间有一条数据通道，主存与 I/O 设备交换数据时可不必通过 CPU，因此数据交换时可省去断点现场保护和现场恢复，工作速度比程序中断方式高。

DMA 与主存交换信息时，主要有三种方法：

1）停止 CPU 访问主存

2）周期挪用（周期窃取）

3）DMA 与 CPU 交替访问

2． DMA 数据传送过程

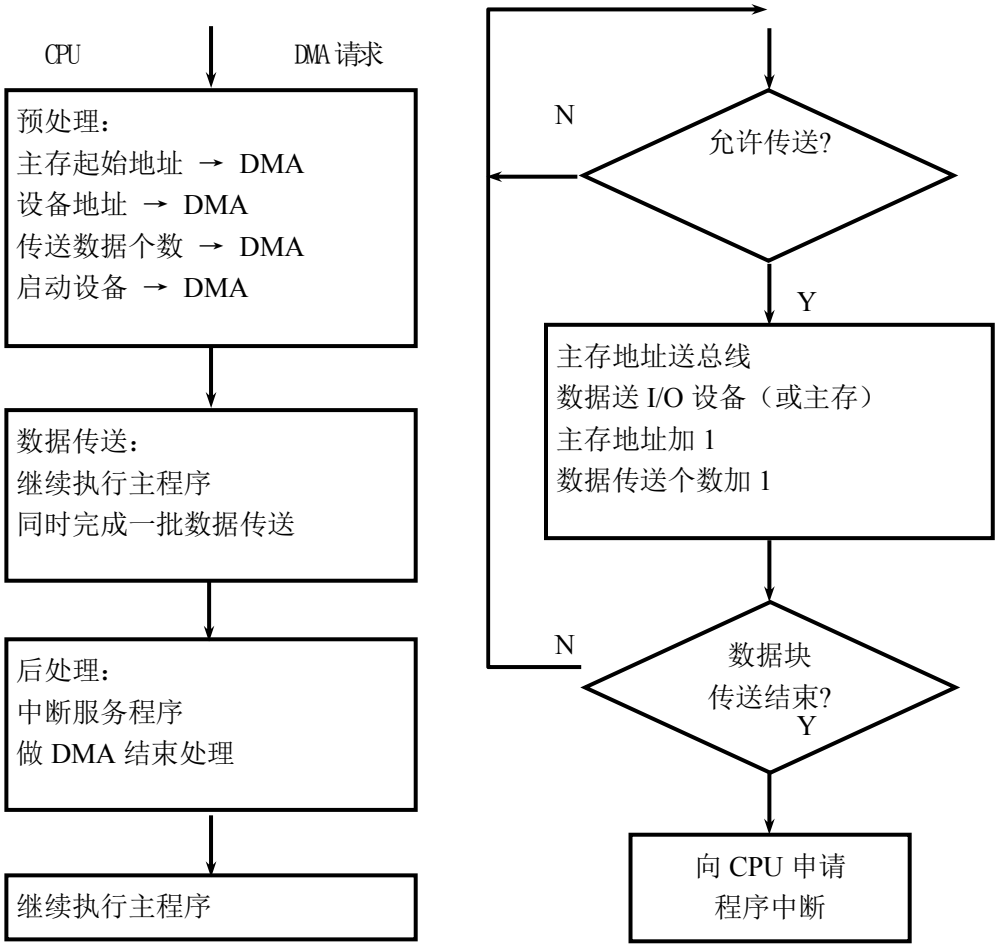
DMA 的数据传送过程分预处理、数据传送和后处理三个阶段。

1）预处理

- 向 DMA 控制逻辑设置数据传输方向（输入/输出）；
- 向 DMA 设备地址寄存器 DR 送入设备号，并启动设备；
- 向 DMA 主存地址寄存器 AR 写入交换数据的起始地址；
- 向 DMA 字计数器 WC 写入交换数据的字数。

2）数据传送

DMA 方式以数据块为单位传送。



(a) DMA 传送　　(b) 数据传送阶段的细化

图 4.16　DMA 传送过程示意图

以周期挪用方式为例，其数据输入的具体操作为：

- 从设备读一个字节到 DMA 的数据缓冲寄存器 BR；
- 设备向 DMA 接口发请求 DREQ；
- DMA 接口向 CPU 申请总线控制权（HRQ）；
- CPU 发回响应信号 HLDA，表示允许将总线控制器交给 DMA 接口；
- 将 DMA 主存地址寄存器中的主存地址送地址总线；
- 通知设备已被授予一个 DMA 周期（DACK）；
- 将 DMA 数据缓冲器的内容送数据总线
- 命令主存储器作写操作；

• 修改主存储器地址和字计数值；

• 判断数据块是否传送结束，若未结束则继续传送；若已结束（字计数器溢出），则向CPU 申请程序中断，标志数据块传送结束。

3）后处理

当 DMA 接口的中断请求得到响应后，CPU 进行某些 DMA 结束工作，如数据校验等。若需要再次传送数据，则重新对 DMA 接口初始化。

4.3.4. I/O 通道和 I/O 处理机介绍

在小型和微型计算机中，采用 DMA 方式可实现高速 I/O 设备与主机之间成组数据的交换，但在大中型计算机中，I/O 设备数量多，数据传送频繁，若仍采用 DMA 方式，则需要为每台 I/O 设备都配置专用的 DMA 接口，硬件成本高，而且存在多个 DMA 接口同时访问主存的冲突问题，CPU 对众多的 DMA 接口进行管理使得控制系统变得很复杂。

1. 通道

通道是用来负责管理 I/O 设备以及实现主存与 I/O 设备之间交换信息的部件，可以视为一种具有特殊功能的处理器。通道有通道专用的输入输出指令，用这些指令可以编制出独立输入输出程序供通道运行。但通道本身不是一个完全独立的处理器。它根据 CPU 的发出的 I/O 指令启动、停止或改变工作状态，是从属于 CPU 的一个专用处理器。依赖通道管理的 I/O 设备在与主机交换信息时，CPU 不直接参与管理，故提高了 CPU 的资源利用率。

2. I/O 处理机

I/O 处理机又称为外围处理机(Peripheral Processor)，它基本独立于主机工作，既可完成 I/O 通道要完成的 I/O 控制，又可完成码制变换、格式处理、数据块检错、纠错等操作。具有 I/O 处理机的输入输出系统与 CPU 工作的并行性更高，这说明输入输出系统相对对主机来说具有更大的独立性。

4.4. I/O 接口

4.4.1. 概述

I/O 接口通常是指主机与外部设备之间设置的集成接口电路芯片。常用的接口芯片有串行接口（RS232C、RS485、USB、I2C、SPI）、无线接口（Bluetooth、IrDA）、网络接口（以太网）、并行接口（Centronics、SCSI、IDE）、显示器视频接口（VGA）。每种接口都需相应的驱动程序（Driver）才能正常工作。

主机与外设之间需要接口的主要原因有：

1）高速的 CPU 和慢速的外设之间的速度匹配，如打印机就是典型的慢速输出设备。

2）完成不同数据格式的转换，如串/并转换。

3）完成不同电气特性的转换匹配，如 RS232C 接口完成 TTL 电平到 RS232C 电平的转换。

4）传递 CPU 和外部设备之间的状态和控制信号。

4.4.2. 接口的分类

I/O 接口按不同的方式可以有以下的分类：

1）按数据传送方式 —— 并行接口和串行接口；

2）按功能选择的灵活性 —— 可编程接口和不可编程接口；

3）按接口通用性 —— 通用接口和专用接口；

4）按数据传送的控制方式 —— 程序控制式接口和 DMA 式接口；

4.4.3. 接口标准

为了方便计算机外部设备制造商生产的设备能方便地与计算机连接，IEEE 和有关制造厂商联盟制定了许多的外部设备接口标准。接口标准对接口的用途、接口信号名称及电气特性、连接器机械特性等均作出了详细的规定，微型计算机和嵌入式系统中常用的外设接口标准见表 4.1。

表 4.1　微型计算机常用 I/O 接口

名 称	数据传输方式	数据传输速率	标 准	插头/插座形式	可连接的设备数目	通常连接的设备
串行口	串行，双向	50～19200 b/s	EIA-232 或 EIA-422	DB25F 或 DB9F	1	鼠标器，MODEM
并行口(增强式)	并行，双向	1.5MB/s	IEEE 1284	DB25M	1	打印机，扫描仪
USB1.0 USB1.1	串行，双向	1.5Mb/s(慢速) 12MB/s(全速)		USB A	最多 127	键盘，鼠标器，数码相机，移动盘等
USB2.0	串行，双向	120MB/s(高速)		USB A	最多 127	外接硬盘，数字视频设备，扫描仪等
FireWire(i.Link)	串行，双向	12.5,25,50MB/s、100MB/s	IEEE 1394a IEEE 1394b		最多 63	数字视频设备

IDE	并行，双向	66MB/s，100MB/s 133MB/s	Ultra ATA/66 Ultra ATA/100 Ultra ATA/133	(E-IDE)	1~4	硬盘，光驱，软驱
显示器输出接口	并行，单向	200～500MB/s	VGA	HDB15	1	显示器
红外线接口(IrDA)	串行，双向	115,000 bps 或 4 Mbps	红外线数据协会	不需要	1	键盘，鼠标器，打印机等

一、EIA-RS-232C

1．介绍

RS-232C 标准（协议）的全称是 EIA-RS-232C 标准，是一种广泛使用的异步串行通信接口标准。其中 EIA(Electronic Industry Association)代表美国电子工业协会，RS（recommeded standard）代表推荐标准，232 是标识号，C 代表 RS232 的最新一次修改（1969）。标准规定了连接器机械特性、信号的电气特性及功能、字符的串行传送帧格式。RS-232-C 标准规定的数据传输速率为 300、600、1200、2400、4800、9600、19200 波特率。

RS-232C 建议发送器最大驱动 2500pF 的电容负载，通信距离受驱动器负载电容和传输速率的限制。例如，采用 150pF/m 的通信电缆时，9600bps 下最大通信距离为 15m；若每米电缆的电容量减小，通信距离可以增加。传输距离短的另一原因是 RS-232 属单端信号传送，存在共地噪声和不能抑制共模干扰等问题，因此一般用于 20m 以内的通信。

RS-232C 标准最初为远程通信连接数据终端设备 DTE(Data Terminal Equipment)与数据通信设备 DCE（Data Communication Equipment)而制定的，定义的信号线较多。但目前它又广泛地被借来用于计算机与终端或外设之间的桌面连接标准，实际使用的信号线常常是标准定义的信号的一个子集。

2．RS-232C 的接口连接器及信号定义

RS-232C 标准接口采用 DB-25 作为标准连接器，DB-25 共有 25 芯，，由于只有 9 根信号常用，又定义的 DB-9 连接器，DB-9 连接方式是 DB-25 的一个子集。DB-25 和 DB-9 如图 4.17 所示。目前，微型计算机中的 COM1/2 口一般采用 DB-9 作为标准连接器。

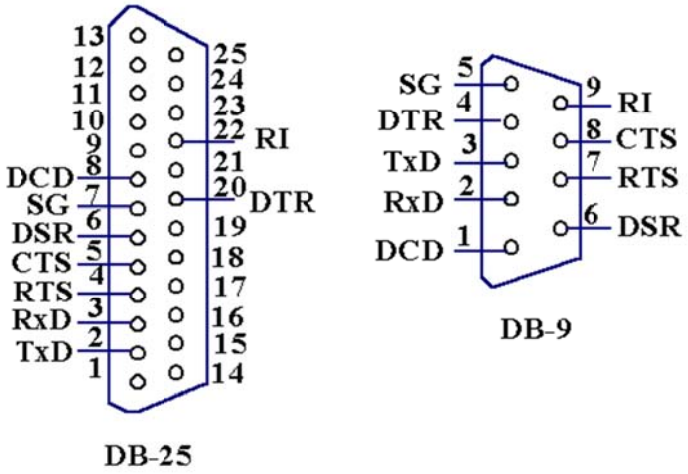


图 4.17　RS-232C 接口使用的 DB-25 和 DB-9 连接器

9 针 RS-232C 串行异步通信的 9 个信号定义如下表所示。

表 4.2 DB-9 各针的信号定义

脚位	名称简写	意　　义
1	CD	载波侦测（Carrier Detect）
2	RXD	接收字符(Receive)
3	TXD	发送字符(Transmit)
4	DTR	数据终端就绪(Data Terminal Ready)
5	GND	地(Ground)
6	DSR	数据准备好(Data Set ready)
7	RTS	要求发送(Request To Send)
8	CTS	清除已发送(Cleat To Send)
9	RI	响铃侦测(Ring Indicator)

3.电气特性

EIA-RS-232C 对接口中各信号的电气特性、逻辑电平都作了规定。

1）数据线（TxD 和 RxD）：

逻辑 1(MARK)=-3V～-15V

逻辑 0(SPACE)=+3～＋15V

2）控制线（RTS、CTS、DSR、DTR 和 DCD）

信号有效（接通，ON 状态，正电压）＝+3V～+15V

信号无效（断开，OFF 状态，负电压)=-3V～-15V

3）字符传送的帧格式

RS-232C 接口设备必须严格按照约定的帧格式发送或接收一个字符。帧格式如下图所示。

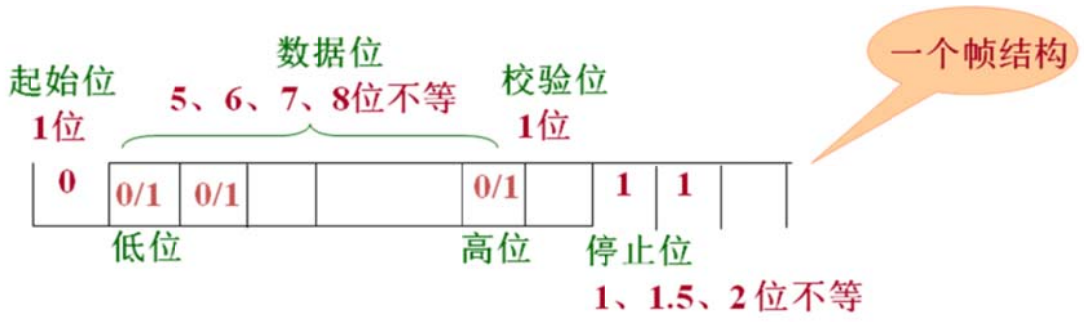


图 4.18　RS-232C 通信中的字符帧格式

一个字符帧包括一个起始位、5-8 位数据位（低位在先，高位在后）、0-1 位奇偶校验位、1-2 位停止位。每位持续时间相同，由通信双方约定的波特率决定。异步串行通信的双方使用各自独立的时钟，收发数据的同步时钟周期实际上就是字符帧中每一位的时间宽度。所以，要求异步串行通信的发送器和接收器使用相同的时钟频率。

4）电平转换

由于 EIA-RS-232C 采用了不同于 TTL 的电平，为了能够同计算机接口或终端的 TTL 器件连接，必须在 RS-232C 与 TTL 电路之间进行电平和逻辑关系的变换。实现这种变换的方法可用分立元件，也可用集成电路芯片。目前较为广泛地使用集成电路转换器件，如 MC1488、SN75150 芯片可完成 TTL 电平到 EIA 电平的转换，而 MC1489、SN75154 可实现 EIA 电平到 TTL 电平的转换。MAX232 芯片可完成 TTL←→EIA 双向电平转换。

二．USB 接口

1．USB 简介

Intel、Compaq、Digital、IBM、Microsoft、NEC、Northern Telecom 等七家世界著名的计算机和通讯公司于 1995 年 11 月正式制定了 USB1.1 通用串行总线（Universal Serial Bus）规范，1999 年推出 USB2.0 规范。

USB1.1 主要应用在中低速外部设备上，它提供的传输速度有低速 1.5Mbps 和全速 12Mbps 两种。1.5Mbps 带宽的 USB 支持如显示器、调制解调器、键盘、鼠标、扫描仪、打印机、光驱、磁带机、软驱等低速设备。12Mbps 带宽的 USB 支持的多媒体设备。USB2.0 向下兼容 USB1.1，数据的传输率将达到 120Mbps～240Mbps，主要支持宽带数字摄像设备及下一代高速扫描仪、打印机及存储设备。

USB 是一种支持热插拔的高速串行传输总线，它使用差分信号来传输数据，最高速度可达 480Mb/S。USB 支持“总线供电”和“自供电”两种供电模式。在总线供电模式下，设备最多可以获得 500mA 的电流。USB2.0 被设计称为向下兼容的模式，当有全速（USB 1.1）或者低速（USB 1.0）设备连接到高速（USB 2.0）主机时，主机可以退化工作在全速或者低速的模式。一条 USB 总线上，可达到的最高传输速度等级由该总线上最慢的“设备”决定，该设备包括主机、HUB 以及 USB 功能设备。

2．USB 体系结构

USB 体系中的设备包括“主机”、“设备”以及“物理连接”三个部分，如图 4.20 所示。其中主机是一个提供 USB 接口及接口管理能力的硬件、软件及固件的复合体，可以是 PC 机，也可以是 OTG 设备。一个 USB 系统中仅有一个 USB 主机；设备包括 USB 功能设备和 USB HUB，最多支持 127 个设备；物理连接即指的是 USB 的传输线。在 USB 2.0 系统中，要求使用屏蔽的双绞线。

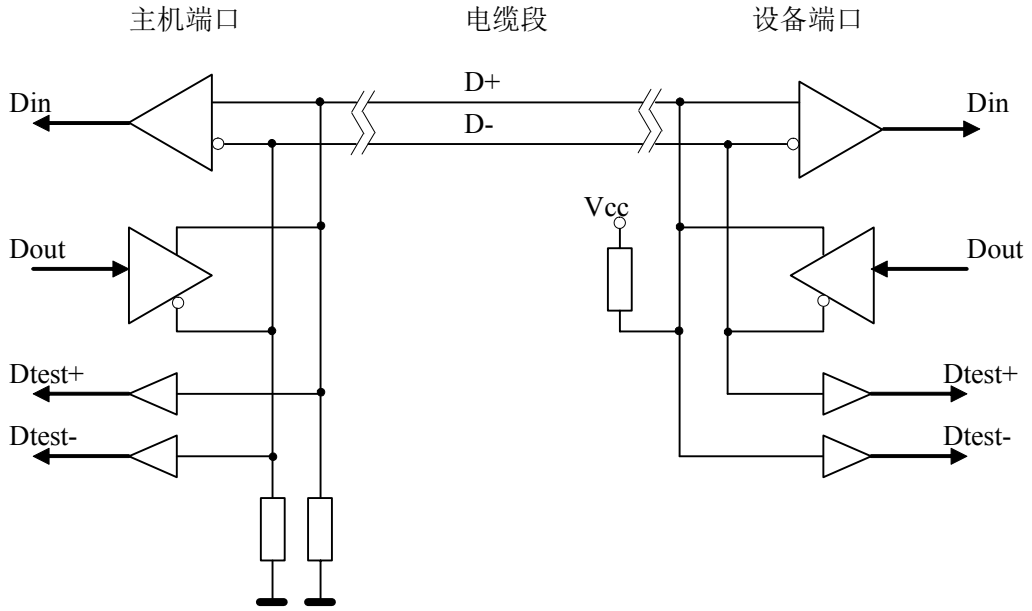


图 4.19　USB 体系中的设备类型

一个 USB HOST 最多可以同时支持 128 个地址，地址 0 作为默认地址，只在设备枚举期间临时使用，而不能被分配给任何一个设备，因此一个 USB HOST 最多可以同时支持 127 个地址，如果一个设备只占用一个地址，那么可最多支持 127 个 USB 设备。在实际的 USB 体系中，如果要连接 127 个 USB 设备，必须要使用 USB HUB，而 USB HUB 也是需要占用地址的，所以实际可支持的 USB 功能设备的数量将小于 127。

三．I2C 接口

I2C 总线（Inter IC BUS）是 Philips 于 1980s 推出的芯片间串行传输总线。它以两根连线（SDA，SCL）实现全双工同步数据传送，可以极方便地构成多机系统和外围器件扩展系统。I2C 总线采用了器件地址的硬件设置方法，通过软件寻址完全避免了器件的片选线寻址方法，从而使硬件系统具有最简单而灵活的扩展方法。目前已被许多公司采用作为 ADC、DAC、E2PROM、RTC 等器件与微处理器互连的方式。

I2C 总线工作原理

采用 I2C 总线系统结构如图 4-20 所示。

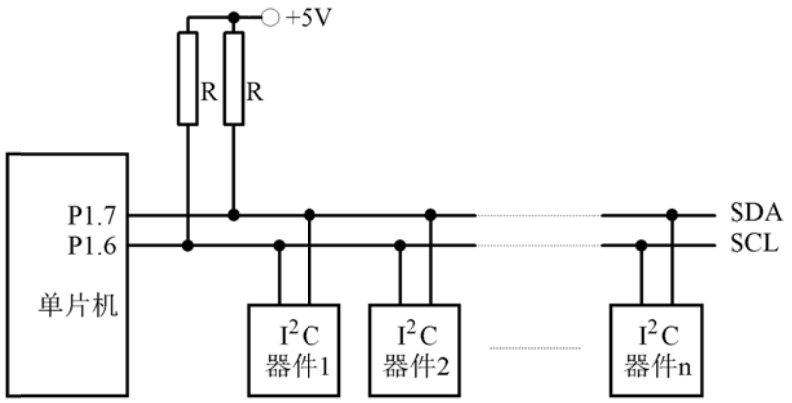


图 4-20 I2C 总线系统结构图

其中，SCL 是时钟线，SDA 是数据线。总线上的各器件都采用漏极开路结构与总线相连，因此，SCL、SDA 均需接上拉电阻，总线在空闭状态下均保持高电平。

I2C 总线支持多主和主从两种工作方式，通常为主从工作方式。在主从工作方式中，系统中只有一个主器件（MCU 或 DSP），总线上其它器件都是具有 I2C 总线的外围从器件。在主从工作方式中，主器件启动数据的发送（发出启动信号），产生时钟信号，发出停止信号。为了实现通信，每个从器件均有唯一一个器件地址，具体地址由 I2C 总线委员会分配。

2．I2C 总线工作方式

图 4-21 为 I2C 总线上进行一次数据传输的通信格式。

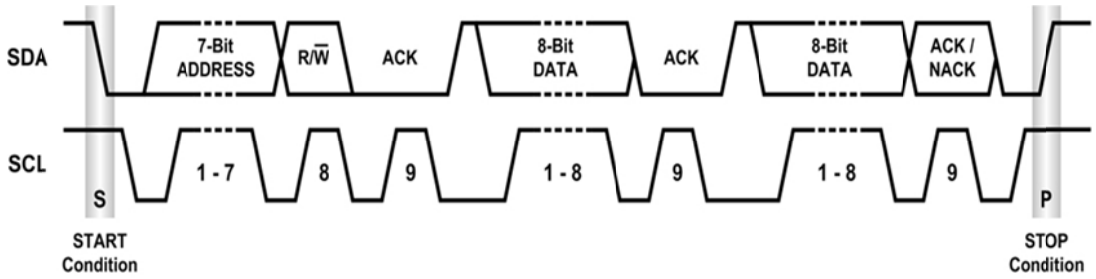


图 4-21　I2C 总线上进行一次数据传输的通信格式

1）．发送启动信号（START）

在利用 I2C 总线进行一次数据传输时，首先由主机发出启动信号启动 I2C 总线。在 SCL 为高电平期间，SDA 出现上升沿则为启动信号。此时具有 I2C 总线接口的从器件会检测到该信号。

2）．发送寻址信号

主机发送启动信号后，再发出寻址信号。寻址信号由一个字节构成，高 7 位为地址位，最低位为方向位，用以表明主机与从器件的数据传送方向。方向位为“0”，表明主机对从器件的写操作；方向位为“1”时，表明主机对从器件的读操作。

3）．应答信号（ACK）

I2C 总线协议规定，每传送一个字节数据（含地址及命令字）后，都要有一个应答信号，以确定数据传送是否正确。应答信号由接收设备产生，在 SCL 信号为高电平期间，接收设

备将 SDA 拉为低电平，表示数据传输正确，产生应答。

4）．数据传输

主机发送寻址信号并得到从器件应答后，便可进行数据传输，每次一个字节，但每次传输都应在得到应答信号后再进行下一字节传送。

5）．非应答信号（NACK）

当主机为接收设备时，主机对最后一个字节不应答，以向发送设备表示数据传送结束。

6）．发送停止信号（STOP）

在全部数据传送完毕后，主机发送停止信号，即在 SCL 为高电平期间，SDA 上产生一上升沿信号。

习题：

1. 什么是输入输出接口？其作用是什么？
2. 微型计算机中，常见的输入输出方式有哪些？
3. 输入输出查询方式有何特点？画出查询方式一般的工作流程？
4. 什么是中断方式？中断的一般响应过程？
5. 什么是 DMA 方式？DMA 工作方式的一般数据传送过程是怎样的？
6. RS-232 串行接口连接器 DB-9F 中定义的信号线有哪些？信号电平是多少？串行通信的字符帧格式如何规定的？影响通信距离的主要因素有哪些？
7. USB 接口信号有哪些？USB 体系中有哪些设备？设备数量有何规定？
8. I2C 接口信号有哪些？I2C 接口有何特点？

第5章. 8086 微机系统原理和结构

微处理器(CPU)是计算机系统的核心部件，控制和协调整个计算机系统的工作，具有以下几项基本功能：

① 能够进行算术运算和逻辑运算；

② 能对指令进行译码、寄存，并执行指令所规定的操作；

③ 具有与存储器和 I/O 接口进行数据通信的能力；

④ 具有少量数据的暂存能力；

⑤ 能够提供这个系统所需的定时和控制信号；

⑥ 能够响应输入／输出设备发出的中断请求。

人们通常用 CPU 内部操作中的数据位数(内部寄存器位数及内部数据总线位数)来作为对其总体性能的一个表征。通常所说的 16 位机、32 位机是表示该计算机中微处理器内部数据总线的宽度，即 CPU 可同时操作的二进制码的位数。微型计算机有 8 位、16 位或 32 位 CPU 等，其含义是可操作 8 位、16 位或 32 位二进制码。目前常用的 CPU 都是 32 位的，即一次可传送 32 位二进制数。

本章 8086 微机系统的一般结构、系统配置、存储器组织、工作时序以及 8086 的指令系统、存储器扩展等内容。

5.1. 8086 CPU 结构与功能

5.1.1. 8086 的结构特点

8086CPU 是典型的 16 位微处理器，即它的内部数据总线、内部寄存器以及外部数据总线都是 16 位的。它是 Intel 公司第三代 CPU，时钟频率为 5 MHz，是具有 40 根引出线的双列直插式芯片，共有 20 位地址线，其中 16 位为数据/地址复用线，可寻址 1 MB 内存单元、64KB 的 I/O 地址空间。其副产品 8088 CPU 除外部数据总线是 8 位的之外，在其他方面的性能几乎完全一样。

以 8086/8088 为处理器的 IBM-PC 个人计算机曾风靡全世界，也使 8086/8088 成为最主要的微处理器结构。8086CPU 具有如下结果特点：

(1) 指令执行的流水线技术

与 Intel 公司第二代 CPU 相比，8086CPU 采用了流水线技术。尽管在这方面还不能与现在新型的 CPU(如 Pentium、K7 等)相比，但由于是它首先在微处理器中采用流水线结构技术的，从而使它成为 CPU 发展史上的一个里程碑。

在程序执行过程中，CPU 有规律地重复执行以下步骤：

① 从存储器中取一条指令；

② 指令译码；

③ 读取操作数(如果需要)；

④ 执行指令；

⑤ 存放结果(如果必要)。

这里，读取操作数和存放结果不是必需的操作，要根据指令的情况来定。必需的操作是取指令、指令译码和指令执行 3 步。其中，取指令由 CPU 内部的总线接口单元(Bus Interface Unit，BIU)完成，而指令译码和执行由执行单元(Execution Unit，EU)实现。8088/8086 CPU 以前的微处理器都是用串行方式完成这些操作的，即依照取指令、译码、执行指令这样的顺序一步一步地执行。CPU 的取指令和执行指令不能并行进行；另外，某些指令的执行并不需要访问内存，所以在指令的执行过程中外部总线也处于空闲状态。工作过程如图 5. 1 (a)所示。这样的结构显然使 CPU 和总线的工作效率都比较低。

从 8086/8088 开始，CPU 采用了一种新的结构来并行地完成这些工作。执行单元负责执行指令，总线接口单元负责取指令、取操作数和写结果。它们独立地、并行地完成各自相应的工作。当 EU 执行指令时，BIU 便“预取”下一条要执行的指令，所以大多数情况下取指令的时间可“省掉”，从而加快了程序的运行速度。这种结果称为流水线结构。流水线操作的示意图如图 5. 1 (b)所示。

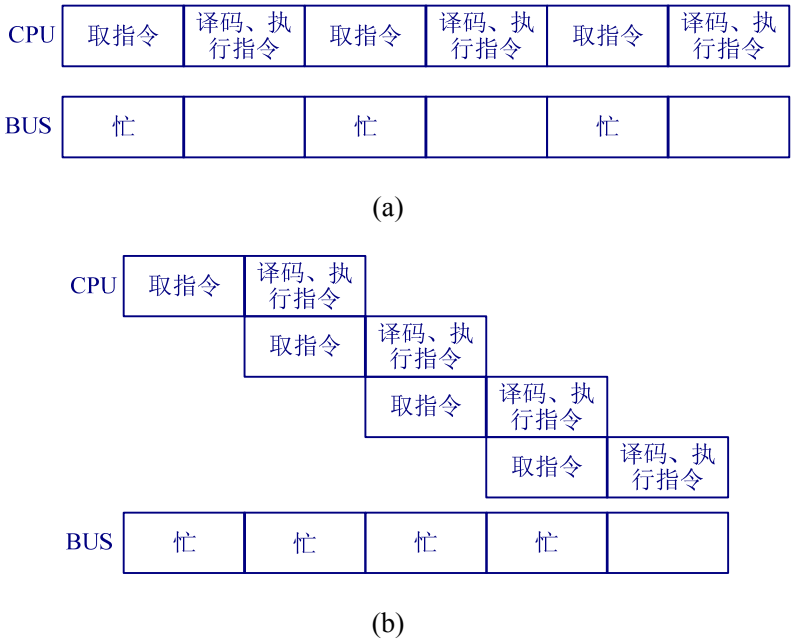


图 5. 1　重叠的取指令和执行指令操作(流水线)

在流水线操作中，因预取的指令有可能不是马上执行，故需要有一个暂时存放所预取的指令的地方。8086 内部设置了一个 6 B(8088 为 4 B)的指令流队列(Instruction Stream Queue)，实际上是一个内部的存储器阵列，存储数据为先进先出。

(2) 存储器的分段结构

8086 的地址总线为 20 位，可寻址 220=1 MB 的内存空间；但其内部寄存器和内部地址总线都只有 16 位，也就是说能够由 ALU 提供的最大地址空间只能是 64 KB。为了实现 CPU 对 1 MB 空间的寻址，8086 将内存储器空间分为若干逻辑段，每个段最大为 64KB，并在 CPU 中专门设置了一些段寄存器，用于存放逻辑段的起始地址，这些起始地址是 16 位的，

满足内部地址总线的宽度要求。

一个实际的存储器单元的地址是由它所在段的段地址和该单元与段起始地址之间的位移量(偏移地址)按一定规律共同构成的一个 20 位的地址，称为物理地址。通过这样的方法，使得 8086 能够实现对 1 MB 空间的管理。有关物理地址的构成方法将在 5.3 节介绍。

(3) 支持用于浮点运算的协处理器及多微处理器系统

在 8086 以前的微处理器中，都是采用定点数据表示来实现浮点运算的，实现的方法大多是通过子程序来实现。这样做很费时，一般要使处理器的速度降低两个数量级。如用一台定点运算速度为1 000万次 / s的计算机进行科学计算，则其实际运算速度将低于10万次/s。不仅如此，CPU 与主存储器之间的数据通信量也会大大增加。

8086CPU 可支持与用于浮点运算的数学协处理器 8087 之间的连接，从而大大地提高系统的运算速度和数据处理能力。

另外，8086CPU 在指令方面和结构设计上，都考虑了支持使用该微处理器构成一个共享总线的多微处理器系统。

5.1.2. 8086 的功能结构

8086 的内部结构如图 5. 2 所示。它由执行单元 EU 和总线接口单元 BIU 两大部分构成。执行单元 EU 负责执行指令。它由算术逻辑单元(ALU)、通用寄存器组、16 位标志寄存器(FLAGS)、EU 控制电路等组成。EU 在工作时直接从指令流队列中取指令代码，对其译码后产生完成指令所需要的控制信息。数据在 ALU 中进行运算，运算结果的特征保留在标志寄存器 FLAGS 中。

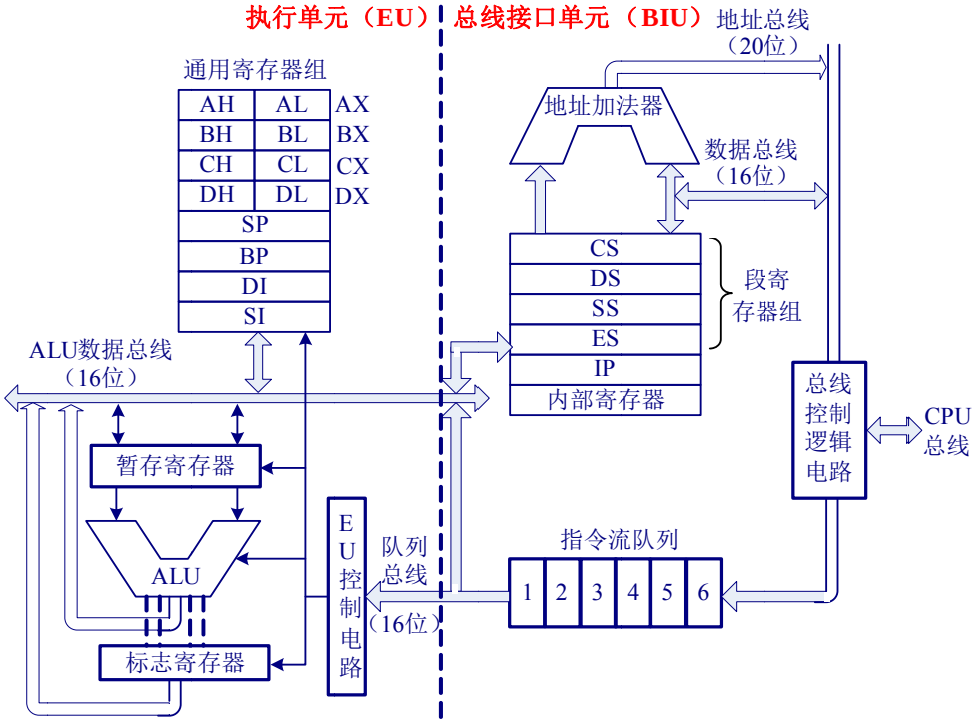


图 5. 2　8086 的内部结构

总线接口单元 BIU 负责 CPU 与存储器和 I/O 接口之间的信息传送。它由段寄存器、指令指针寄存器、指令流队列、地址加法器以及总线控制逻辑组成。8086CPU 的指令队列长度为 6B。

当 EU 取走指令、指令流队列出现 2 个以上空单元时，BIU 就自动执行一次取指令周期，从内存中取出后续的指令代码放人队列中；如果 EU 需要数据，BIU 会根据 EU 给出的地址，从指定的内存单元或外设中取出数据供 EU 使用；运算结束后，BIU 负责将运算结果送入指定的内存单元或外设。如果指令流队列为空，EU 就等待，直到有指令为止。若 BIU 正在取指令时 EU 发出访问总线的请求，则必须等 BIU 取指令完毕后，该请求才能得到响应。一般情况下，程序是顺序执行的，而当遇到跳转指令时，BIU 使指令队列复位，从新地址取出指令，并立即传给 EU 去执行。

指令流队列的存在使 8086CPU 的 EU 和 BIU 并行工作，从而减少了 CPU 为取指令而等持的时间，提高了 CPU 的利用率，加快了整机的运行速度。另外也降低了对存储器存取速度的要求。

5.1.3. 8086 的寄存器组

8086CPU 内部共有 14 个 16 位寄存器。按其功能可分为三大类，即通用寄存器组、段寄存器组以及控制寄存器组，其结构如图 5. 3 所示。

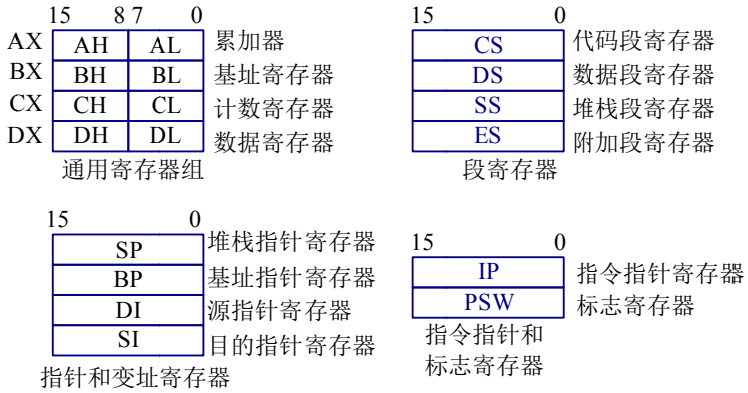


图 5. 3　8086 内部寄存器结构

1. 通用寄存器组

通用寄存器组包括 4 个数据寄存器、2 个地址指针寄存器和 2 个变址寄存器。

(1)　数据寄存器 AX、BX、CX、DX

数据寄存器一般用于存放参与运算的数据或运算的结果。每一个数据寄存器都是 16 位寄存器，但又可将高 8 位和低 8 位分别作为两个独立的 8 位寄存器使用。它们的高 8 位记为 AH、BH、CH、DH，低 8 位记为 AL、BL、CL、DL。这种灵活的使用方法给编程带来极大的方便，既可以处理 16 位数据，也能处理 8 位数据。

数据寄存器除了作为通用寄存器使用外，它们还有各自特定的用法：

① AX(Accumulator)称为累加器，常用于存放算术逻辑运算中的操作数。所有的 I/O 指令都使用累加器与外设接口传送信息。

② BX(Base)称为基地址寄存器，常用来存放访问内存时的基地址。

③ CX(Count)称为计数寄存器，在循环和串操作指令中用做计数器。

④ DX(Data)称为数据寄存器，在寄存器间接寻址的 I/O 指令中存放 I/O 端口的地址。

另外，在做双字长乘、除法运算时，DX 与 AX 合起来存放一个双字长数(32 位)，其中 DX 存放高 16 位，AX 存放低 16 位。

(2) 地址指针寄存器 SP、BP

SP(Stack Pointer)称为堆栈指针寄存器，它在堆栈操作中用来存放栈顶的偏移地址，永远指向堆栈的栈顶。

BP(Base Pointer)称为基地址指针寄存器。一般也常用来存放访问内存时的基地址。但它通常是与 SS 寄存器配对使用(BX 通常是与 DS 寄存器配对使用)。

作为通用寄存器，SP 和 BP 也可以存放数据。但实际上，它们更重要的用途是存放内存单元的偏移地址，特别是在访问堆栈时作为指向堆栈的指针(因为它们默认的段寄存器都是堆栈段寄存器 SS)。

(3) 变址寄存器 SI、DI

SI(Source Index)称为源变址寄存器，DI(Destination Index)称为目标变址寄存器，它们常常在变址寻址方式中作为索引指针。在字符串操作指令中，要求用 SI 作为源变址寄存器，存放源操作数的偏移地址；DI 作为目标变址寄存器，存放目标操作数的偏移地址。

以上 8 个 16 位寄存器作为通用寄存器都具有相同的功能，即存放操作数及运算结果，除此之外，在不同的场合它们各自又有自己独特的用法。

2. 段寄存器组

8086 的存储器是分段管理的。每一个存储单元的地址都是由它所在逻辑段的段基地址和段内的偏移地址两部分构成。CPU 在访问存储器时，首先要找到访问单元所在的段，也就是确定单元所在的段基地址。段寄存器就是用来存放段基地址的，即逻辑段起始地址的高 16 位。8086 共有 4 个 16 位段寄存器，分别是代码段寄存器 CS(Code Segment)、数据段寄存器 DS(Data Segment)、附加段寄存器 ES(Extra Segment)和堆栈段寄存器 SS(Stack Segment)。

①代码段寄存器 CS　代码段存放的是当前执行程序的指令代码。CS 的内容是代码段的段基地址，它和指令指针 IP 一起决定下一条所要执行指令的物理存储地址。

②数据段寄存器 DS　数据段通常用来存放数据和字符。DS 存放当前数据段的段基地址。

③附加段寄存器 ES　附加段是一个附加数据段，主要用在字符串操作时作为目标地址使用。ES 的内容就是加段的段基地址。

④堆栈段寄存器 SS　堆栈是在存储器中开辟的一个特殊存储区，用于存放当前暂时不用但又需要保存的数据和地址。如在子程序调用或响应中断时需要保存返回主程序的地址和进入子程序后将要改变其值的寄存器的内容。堆栈的操作遵循先进后出的原则，操作的地址由 SS(堆栈段基地址寄存器)和 SP(堆栈指针寄存器)的内容指定。

3. 控制寄存器

8086 内部包括指令指针 IP 和标志寄存器 FLAGS 两个控制寄存器。

(1) 指令指针四(Instruction Pointer)

IP 用来存放下一条要执行指令的偏移地址。CPU 取指令时总是以 CS 的内容为段基地址，以 IP 为段内偏移地址。当 CPU 从 CS 段偏移地址为(IP)的内存单元中取出指令代码的一个字节后，IP 自动加 1，指向指令代码的下一个字节。如果遇到过程调用、转移及返回等指令时，系统将根据程序确定新的 IP 的内容，使其不再加 1。

用户程序不能直接访问 IP，即指令中的操作数不能是 IP。

(2) 标志寄存器 FLAGS

FLAGS 称为标志寄存器或程序状态字(PSW)，是一个 16 位寄存器，但只使用了其中的 9 位，包括 6 个状态标志位和 3 个控制标志位，如图 5.4 所示。

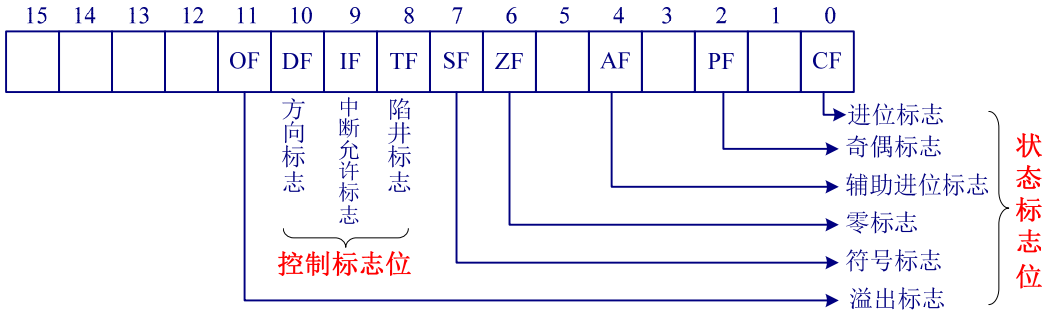


图 5.4　8086 的标志寄存器

1) 状态标志位

状态标志位记录了算术和逻辑运算结果的一些特征。例如，结果是否为 0、是否有进位或借位、结果是否溢出等。不同的指令对状态标志位具有不同的影响。

CF(Carry Flag)　进位标志位。在进行加(减)法运算时，若最高位向更高位有进(借)位时 CF=1，否则 CF=0。

PF(Parity Flag)　奇偶标志位。当运算结果的低 8 位中“1”的个数为偶数时 PF=1，为奇数时 PF=0。

AF(Auxiliary Carry Flag)　辅助进位标志位。在加(减)法操作中，若 b3 向 b4 有进位(借位)时 AF=1，否则 AF=0。用 DAA 和 DAS 指令可测试这个标志位，以便在 BCD 加法或减法之后调整 AL 中的值。

ZF(Zero Flag)　零标志位。当运算结果各位均为零时 ZF=1，否则 ZF=0。

SP(Sign Flag)　符号标志位。当运算结果的最高位(字节操作时是 b7，字操作时是 bl5)为 1 时 SF=1，否则 SF=0。即它和运算结果的最高位相同。因为在补码运算时最高位为符号位，SF=1 表示结果为负；SF=0 表示结果为正。

OF(Over flow Flag)　溢出标志位。当算术运算的结果超出了带符号数的范围，即溢出时 OF=1，否则 OF=0。8 位带符号数的范围是-128~+127，16 位带符号数的范围是-32 768~+32 767。

下面以一个简单的算术运算例来说明操作结果对状态标志位的影响。

假设执行一条加法指令，计算 5394H-777FH 后各状态标志位的状态为何?

解 采用补码运算，5394H-777FH=5394H 补+8881H 补

```
  0101 0011 1001 0100
+ 1000 1000 1000 0001
-------------------
  1101 1100 0001 0101
```

运算结果为 DC15H 补=-23EBH，则执行这条加法指令后标志寄存器的状态为：CF=0，PP=0，AF=0，ZF=0，SF=1，OF=0。

2) 控制标志位

控制标志位用于设置控制条件。控制标志被设置后便对其后的操作产生控制作用。

TF(Trap Flag) 陷井标志位，也称为跟踪标志位。TF=1 时，CPU 处于单步执行指令的工作方式。这种方式便于进行程序的调试，每执行一条指令，自动产生一次单步中断，从而使用户能逐条地检查指令。

IF(Interrupt Enable Flag) 中断允许标志位。IP=1 时，CPU 可以响应可屏蔽中断请求；IF=0 时，则 CPU 禁止响应可屏蔽中断请求。IP 的状态对不可屏蔽中断及内部中断没有影响。

DF(Direction Flag) 方向标志位。DF=1 时，串操作按减地址方式进行，也就是说，从高地址开始，每处理一个元素，地址指针就自动减 1(或减 2)；DF=0 时，串操作按增地址方式进行。

5.2. 8086 CPU 的引脚及其总线结构

5.2.1. 8086 的引脚定义及其功能

8086CPU 是具有 40 条引出线的、采用双列直插式封装的集成电路芯片，其外部引脚如图 5.5 所示。图 5.6 给出了 8086CPU 内部各功能部件连接的框图。为减少芯片的引脚数，以减少体积，8086CPU 的许多引脚都具有双重定义和功能，采用分时复用方式工作，即在不同时刻，引脚上的信号具有不同的意义。引脚定义的方法大致可分为这样几种：

① 每根引脚只传送一种信息，如读信号线 $\overline{RD}$ 。

② 引脚电平的高低代表不同的信号，如 $M/\overline{IO}$，在为低电平时，表示当前访问的是存储器；为高电平时，则表示访问输入 / 输出接口。

③ 在不同的时间范围，引脚传送不同的信息，即分时复用。如 AD0~ADl5，在某一时刻传送地址的低 16 位信号；另一时刻则传送 16 位数据。

④ 在引脚作为输入端或输出端时传送不同的信息。如 $\overline{RQ}/\overline{GT_0}$ 端，作为输入端时，输入的是总线请求信号；为输出端时，则输出总线请求允许信号。

⑤ 当 CPU 工作在不同模式时，引脚具有不同的名称和定义。8086 微处理器有两种工作模式，最大模式和最小模式。两种工作模式下的引脚定义有一些区别，图 5.5 所示右边括号中的引脚名称就是 CPU 工作在最大模式时对应引脚的含义。如第 29 根引脚，在最小模式下的定义是写信号 $\overline{WR}$，而在最大模式下的定义则为总线封锁信号 $\overline{LOCK}$。

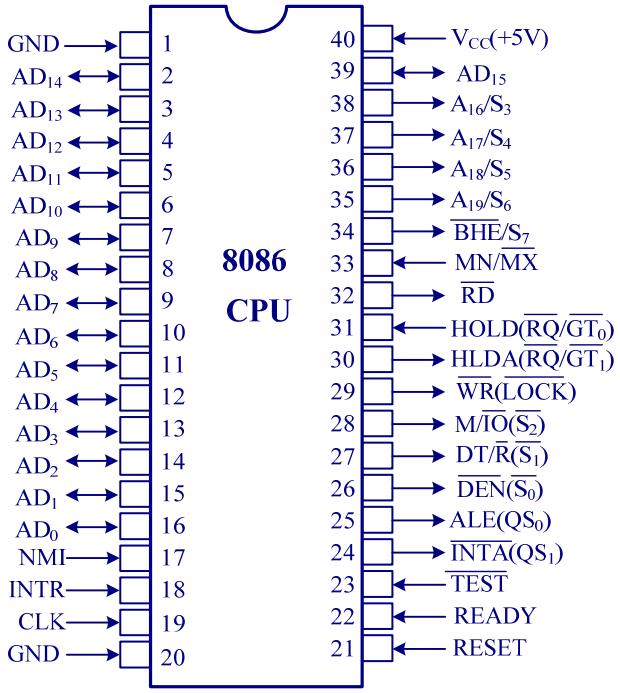


图 5. 5　8086 微处理器芯片引脚图

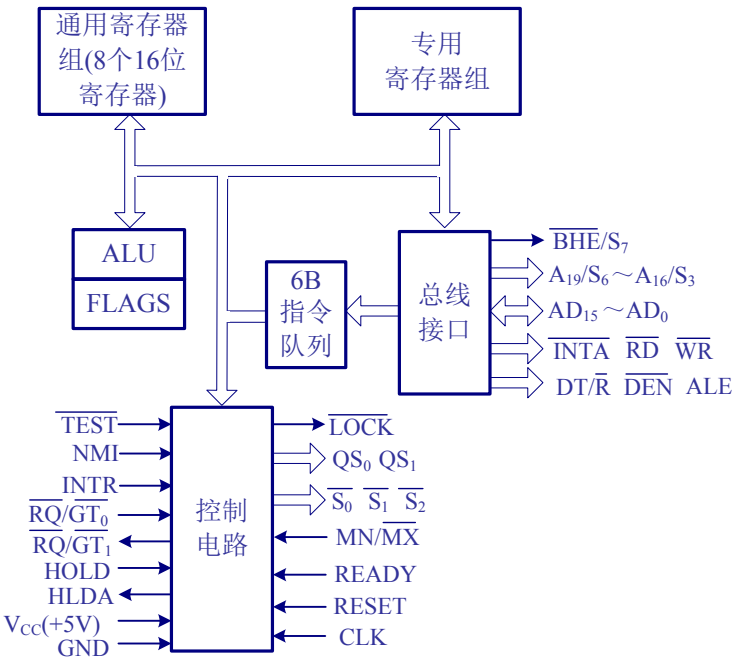


图 5. 6　8086CPU 内部功能模块框图

1. 8086 在两种工作模式下公用引脚的定义

1) 地址／数据总线

8086 有 20 位地址线，16 位数据线，采用分时复用方式，共同占用 20 根引脚。

AD0~ADl5　地址、数据分时复用的双向信号线，三态。当 ALE=1 时，这些引脚上传输的是地址信号；当 $\overline{DEN}$ =0 时，这些引脚上传输的是数据信号。

A16~Al9/S3~S6　分时复用的地址/状态信号线，三态输出。在 8086 访问存储器时，读/写总线周期的第一个机器周期 T1，从这 4 个引脚上送出最高 4 位地址 A16~Al9。而在总线周期的其他机器周期，这 4 个引脚送出状态信号 S3~S6。这些状态信号里，S6 始终为低电平；S5 指示标志寄存器的中断允许标志位 IP 的状态；S4、S3 的组合指示 CPU 当前正在使用的段寄存器，其编码见表 5.1。

表 5.1 S3 和 S4 的功能

S4	S3	当前正在使用的段寄存器
0	0	ES
0	1	SS
1	0	CS 或未使用任何段寄存器
1	1	DS

当 CPU 访问 I／0 端口时，这 4 根引脚不使用，全部为低电平。

2)控制总线

控制总线引脚共有 16 根，这里先讨论两种工作模式共用的 8 根引脚。

MN／$\overline{MX}$　工作方式控制输入。为高电平时，表示 CPU 工作在最小模式；为低电平时，表示 CPU 处于最大模式。

$\overline{RD}$　读选通信号，三态，低电平有效。当其有效时，表示 CPU 正在对存储器或 I/O 接口进行读操作。

READY　“准备好”信号输入引脚，高电子有效。它是由被访问的内存或 I/O 设备发出的响应信号，当其有效时，表示存储器或 I/O 设备已准备好，CPU 可以进行数据传送。

若存储器或 I/O 设备未准备好，则 READY 信号为低电平。CPU 在 T3 周期采样 READY 信号，若其为低电平，CPU 自动插入等待周期 Tw(1 个或多个)，直到 READY 变为高电平后，CPU 才脱离等待状态，完成数据传送过程。

INTR　可屏蔽中断请求输入信号，高电子有效。CPU 在每条指令的最后一个周期采样该信号，以决定是否进入中断响应周期。这个引脚上的中断请求信号可用软件屏蔽。

$\overline{TEST}$　测试信号输入引脚，低电平有效。当 CPU 执行 WAIT 指令时，每隔 5 个时钟

周期对此引脚进行一次测试。若为高电平，CPU 则继续处于空转状态进行等待，直到 $\overline{TEST}$ 引脚变为低电平后，CPU 才结束等待状态，继续执行下一条指令。

NMI　非屏蔽中断请求输入信号，上升沿触发。这个引脚上的中断请求信号不能用软件屏蔽，CPU 在当前指令执行结束后就进人中断过程。

RESET　系统复位输入信号，高电平有效。为使 CPU 完成内部复位过程，该信号至少要在 4 个时钟周期内保持有效。复位后 CPU 内部寄存器的状态见表 5.2。当 RESET 返回低电平时，CPU 将重新启动。

表 5.2 初始化操作

标志寄存器 FR	0000H
指令寄存器 IP	0000H
代码段寄存器 CS	FFFFH
数据段寄存器 DS	0000H
堆栈段寄存器 SS	0000H
附加段寄存器 ES	0000H
指令队列	空

$\overline{BHE}/S_7$　分时复用的控制 / 状态信号线，三态输出。在总线周期的第一个时钟周期输出 $\overline{BHE}$ 信号，其他时钟周期输出状态信号 S7(S7 的意义目前没有定义)。$\overline{BHE}$ 信号的意义是：当 $\overline{BHE}$ 为低电平时，表示可使用高 8 位数据线 AD8~ADl5；否则只使用低 8 位数据线 AD0~AD7。

$\overline{BHE}$ 信号和地址信号一样需要锁存，它同最低位地址信号 A0 的状态组合在一起表示的功能见表 5.3。

表 5.3 $\overline{BHE}$ 与地址信号 A0 的状态组合功能

操作	$\overline{BHE}$	A0	使用的数据线
读或写偶地址的一个字	0	0	AD15~AD0
读或写偶地址的一个字节	1	0	AD7~AD0
读或写奇地址的一个字	0	1	AD15~AD8
	1	1	无效

读或写奇地址的一个字	0	1	AD7~AD0 (第一个总线周期放入低 8 位数据)
	1	0	AD15~AD8 (第二个总线周期放入高 8 位数据)

2. 8086 的两种工作模式及其引脚定义

8086 具有两种工作模式：最小模式和最大模式。最小模式又称为单微处理器模式，在这种模式下，CPU 仅支持由少量设备组成的单微处理器系统而不支持多处理器结构，小系统所需要的全部控制信号都由 CPU 直接提供。对应地，最大模式又称为多微处理机模式。在最大模式下，系统中除了有 8086 CPU 之外，还可以接另外的处理器(如 8087 数学协处理器)，构成多微处理器系统。此时 CPU 不直接提供读／写命令等控制信号，而是将当前要执行的传送操作类型编码成 3 个状态位输出，由总线控制器对状态信号进行译码后产生相应控制信号。其他的控制引脚则直接提供最大模式系统所需要的控制信号。

两种工作模式可以通过在 $MN/\overline{MX}$ 输入引脚加上不同的电平来进行选择。当 $MN/\overline{MX}$ =1 时，8086 工作在最小模式；当 $MN/\overline{MX}$ =0 时，8086 工作在最大模式。两种工作模式下的部分引脚具有不同的功能。

1)最小模式下的引线

引脚 24~31 在最小模式下的功能定义为：

$\overline{INTA}$ 中断响应输出端。当 CPU 响应从 $\overline{INTA}$ 端输入的中断请求时，由 $\overline{INTA}$ 端输出两个连续的负脉冲，可用做外部中断源的中断向量码的读选通信号。

ALE 地址锁存允许信号，三态输出，高电平有效。当它为高电平时，表明 CPU 地址线上有有效地址。可利用它的下降沿将地址信号 A0~A19 和 $\overline{BHE}$ 信号锁存到地址锁存器中。

$\overline{DEN}$ 数据允许信号，三态，低电平有效。该信号有效时，表示数据总线上有有效数据。它在每次访问内存或 I/O 接口以及在中断响应期间有效。它常用做数据总线驱动器的片选信号。

$DT/\overline{R}$ 数据传送方向控制信号，三态。用于确定数据传送的方向。高电平时，CPU 向存储器或 I/O 端口发送数据；低电平时，CPU 从存储器或 I/O 接口接收数据。此信号用于控制总线收发器的传送方向。

$M/\overline{IO}$ 输入／输出／存储器控制信号，三态。用来区分当前操作是访问存储器还是访问 I/O 端口。引脚输出为高电平时，表示访问存储器；为低电平时，则表示访问 I/O 端口。

$\overline{WR}$　写信号输出，三态。此引脚输出为低电平时，表示 CPU 正在对存储器或 I/O 端口进行写操作。

HOLD　总线保持请求信号输入，高电平有效。当某一总线主控设备要占用系统总线时，通过此引脚向 CPU 提出请求。

HLDA　总线保持响应信号输出，高电平有效。这是 CPU 对 HOLD 请求的响应信号，当 CPU 收到有效的 HOLD 信号后，就会对其做出响应：一方面使 CPU 的所有三态输出的地址信号、数据信号和相应的控制信号变为高阻状态(浮动状态)；同时还输出一个有效的 HLDA，表示处理器现在已放弃对总线的控制。当 CPU 检测到 HOLD 信号变低电平后，就立即使 HLDA 变低电平，同时恢复对总线的控制。

在 8086 最小模式下，由 $M/\overline{IO}$、$\overline{RD}$ 和 $\overline{WR}$ 的组合决定了当前数据传送的类型。其组合状态见表 5.4。

表 5.4 $M/\overline{IO}$、$\overline{RD}$ 和 $\overline{WR}$ 的组合决定的数据传送类型

$M/\overline{IO}$	$\overline{RD}$	$\overline{WR}$	传送类型
0	0	1	读 I / O 端口
0	1	0	写 I / O 端口
1	0	1	读存储器
1	1	0	写存储器

2) 最大模式下的引线

当 $MN/\overline{MX}$ 加上低电平时，8086 工作在最大模式下。此时，引脚 24~34 的信号功能为图 3.6 中括号内所示。

QSl、QS0　指令流队列状态输出。如前所述，8086 在执行当前指令的同时，从存储器预取后边的指令放在指令流队列中。QSl、QS0 提供指令流队列的状态信号，以便外部逻辑跟踪 CPU 内部的指令流队列。QSl、QS0 表示的状态见表 5.5。

表 5.5 QSl、QS0 的组合及对应的操作

QSl	QS0	操　　作
0	0	无操作
0	1	队列中操作码的第一个字节
1	0	队列空
1	1	队列中非第一个操作码字节

$\overline{S_2}$、$\overline{S_1}$、$\overline{S_0}$ 总线周期状态信号输出，低电平有效，三态。这 3 个信号连接到总线控制器的输入端，译码后可以产生系统总线所需要的各种控制信号。$\overline{S_2}$、$\overline{S_1}$、$\overline{S_0}$ 的代码组合以及对应的操作见表 5.6。

表 5.6 $\overline{S_2}$、$\overline{S_1}$、$\overline{S_0}$ 的组合及对应的操作

$\overline{S_2}$ $\overline{S_1}$ $\overline{S_0}$	对应的操作
0 0 0	发中断响应信号
0 0 1	读 I / O 端口
0 1 0	写 I / O 端口
0 1 1	暂停
1 0 0	取指令
1 0 1	读存储器
1 1 0	写存储器
1 1 1	无作用

$\overline{LOCK}$ 总线封锁信号输出，低电平有效。该信号有效时，CPU 锁定总线，不允许其他的总线控制设备申请使用系统总线。$\overline{LOCK}$ 信号由前缀指令“LOCK”产生，并维持到 LOCK 指令后面的一条指令执行完为止。另外，CPU 在 INTR 端输入的中断请求信号也会使 $\overline{LOCK}$ 端从第一个 $\overline{INTA}$ 脉冲起到第二个 $\overline{INTA}$ 脉冲结束都保持低电平，以保证在中断响应周期之后，才允许其他总线控制设备使用总线。

$\overline{RQ}/\overline{GT_1}$、$\overline{RQ}/\overline{GT_0}$ 总线请求 / 总线响应信号引脚。每一个引脚都具有双向功能，既是总线请求输入，也是总线响应输出；但是 $\overline{RQ}/\overline{GT_1}$ 比 $\overline{RQ}/\overline{GT_0}$ 具有更高的优先权。这些引脚内部都有上拉电阻，所以在不使用时可以悬空。这两个引脚的功能如下：

当其他的总线控制设备要使用系统总线时，就会产生一个总线请求信号(一个时钟周期宽的负脉冲)，并把它送到 $\overline{RQ}/\overline{GT_1}$ (或 $\overline{RQ}/\overline{GT_0}$)引脚，类似于最小模式下的 HOLD 信号。CPU 检测到总线请求信号后，在下一个 T4 或 T1 期间，在 $\overline{RQ}/\overline{GT_1}$ (或 $\overline{RQ}/\overline{GT_0}$)引脚送出总线响

应信号(一个时钟周期宽的负脉冲)给请求总线的设备，它类似于最小模式下的 HLDA 信号。然后从下一个时钟周期开始，CPU 释放总线。总线请求设备使用完总线后，再产生一个 $\overline{RQ}/\overline{GT_1}$ (或 $\overline{RQ}/\overline{GT_0}$)信号。CPU 检测到该信号后，从下一个时钟周期开始重新控制总线。

此外，在最大模式下，$\overline{RD}$ 引脚不再使用。

5.2.2. 8086 的总线结构

1. 最小模式下的总线结构

这种模式下的 8086 系统构成如图 5.7 所示。图中，CPU 的 20 根地址信号线通过 3 片 8282(或 74L8373)锁存器与系统的地址总线相连。16 位数据线通过一片 8286(或 74LS245)双向总线驱动器连接到系统的数据总线上。因最小模式为单处理器模式，小系统所需的全部控制信号由 CPU 直接产生，可直接接人总线。这样，就构成了一个最小模式下的 8086 系统。在实际的系统中，还应考虑以下两个问题：

① 系统总线的控制信号是 8086CPU 直接产生的。若 8086 CPU 驱动能力不够，可以加总线驱动器 74LS244，以提高驱动能力。

② 按此构成的系统尚不能进行 DMA(Direct Memory Access)传送，因为未对系统总线形成器件(8282、8286)做进一步控制。

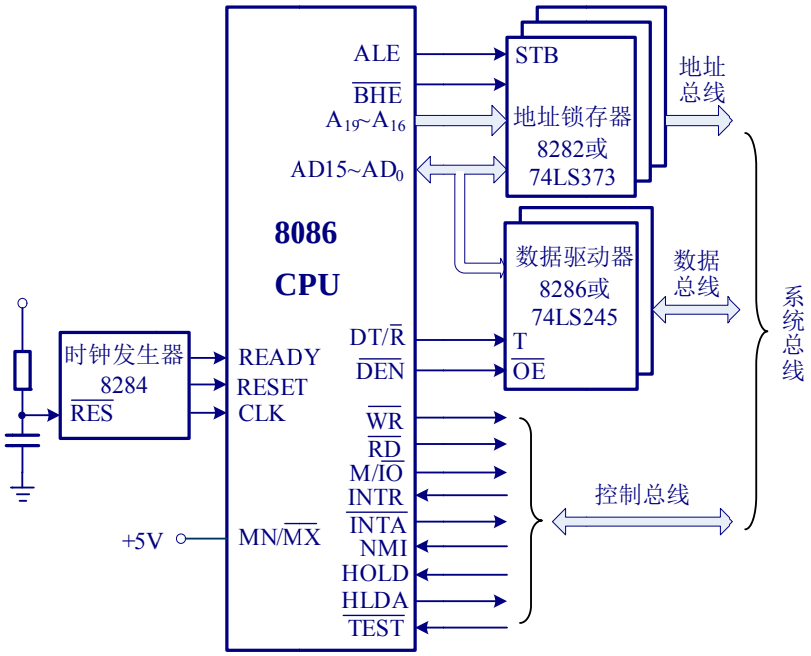


图 5. 7　最小模式下的 8086 系统构成示意图

2. 最大模式下的总线结构

最大模式是支持多处理器的工作模式，即系统中可以有两个或多个能同时执行指令的处理器，增加的处理器可以是 8086 CPU，也可以是数字协处理器 8087 或 I / O 处理器 8089 等。IBMPC 系列微型计算机中的处理器就工作在最大模式，系统中配置了数字协处理器 8087，

提高了系统的数据处理(特别是浮点数的处理)能力。

与单处理器系统不同的是，在多处理器系统中可能会存在多个处理器同时需要使用总线、处理器间的信息传递等问题。因此需要增加专门的控制逻辑电路来确保每次只有一个处理器占用总线，确保一个处理器能够准确地将任务分配给另一个处理器或从另一个处理器中取回结果。

为支持多处理器结构，当工作在最大模式下时，CPU 通过总线控制器 8288 与系统的控制总线相连，由总线控制器提供所有总线控制信号和命令信号。其系统总线结构如图 5.8 所示。图中的 8282 和 8286 也可分别用 74LS373 和 74LS245 代替。当然，这里同样没有考虑在系统总线上实现 DMA 传送的问题。在进行 DMA 传送时，需要保证总线形成电路所有输出信号都呈现高阻状态，即放弃对系统总线的控制。

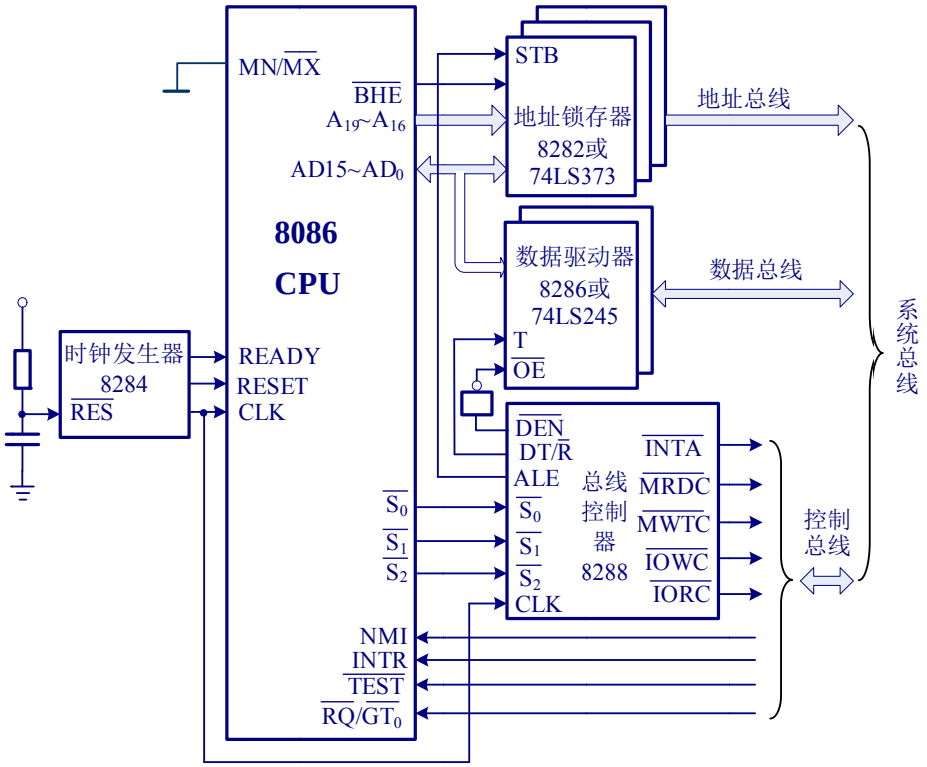


图 5. 8　最大模式下的 8086 系统构成示意图

5.3. 8086 存储器组织

8086 可寻址 1 MB 的存储器空间。要做到这一点，最基本的是每个空间要有惟一的地址，即地址编码至少要是 20 位二进制数(220=1 MB)。本节介绍 20 位地址的形成及存储器中的分段组织。

5.3.1. 存储器地址的分段

1. 存储器地址的分段

在 8086 的存储器中是以字节为单位存储信息的，每个存储单元有唯一的地址来确定。8086 系统有 20 根地址线可寻址 1MB 字节的存储空间，即对存储器寻址要 20 位物理地址，

而 8086 为 16 位机，CPU 内部寄存器只有 16 位，可寻址 64KB。因此 8086 系统把整个存储空间分成许多逻辑段，每段容量不超过 64KB。8086 系统对存储器的分段采用灵活的方法，允许各个逻辑段在整个存储空间中浮动，这样在程序设计时可使程序保持相对的完整性。段和段之间可以是连续的（整个存储空间分成 16 个逻辑段），也可以是分开的或重叠的。如图 5.9 所示。

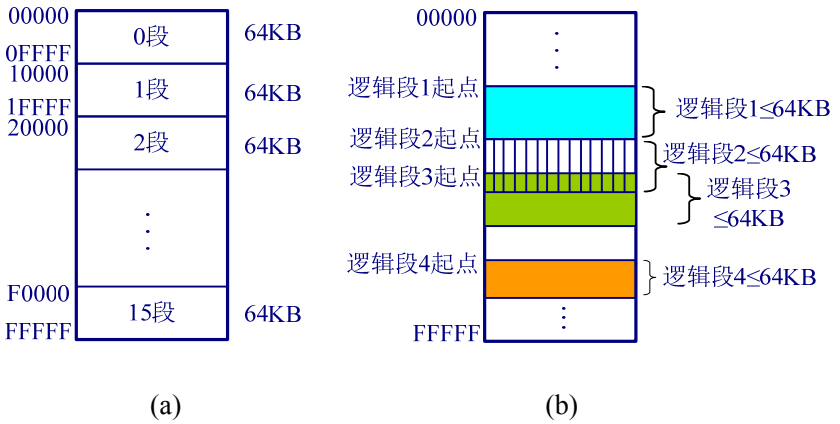


图 5. 9　存储器地址

8086 系统的任何一个存储单元的实际地址，都是由段地址及段内偏移地址两部分组成，从图 5.9 可以看出，一个存储单元，可以在一个段中定义，也可定义在两个重叠的逻辑段中，关键是看段的首地址如何指定。

2．物理地址形成

8086 系统将段地址放在段寄存器中，称为“段基址”。有 4 个段寄存器，分别为代码段寄存器 CS，数据段寄存器 DS，附加段寄存器 ES 和堆栈段寄存器 SS。

段内“偏移地址”指出了从段地址开始的相对偏移位置。它可以放在指令指针寄存器 IP 中，或 16 位通用寄存器中。

由段基址和偏移地址组合而成逻辑地址，程序设计时采用逻辑地址。例如，CS=1000H，IP=200H，则由 CS 和 IP 构成代码段的逻辑地址，记为 CS:IP=1000H:200H。

利用逻辑地址，可以得到 CPU 访问存储器的实际寻址地址，其计算方法为

物理地址=段基址×16+偏移地址

因为段基址指每段的起始地址，它必须是每小段的首地址，其低 4 位一定为 0，所以在实际工作时，是从段寄存器中取出段基址，将其左移 4 位，再与 16 位偏移地址相加，就得到了物理地址，此地址在 CPU 的总线接口部件 BIU 的地址加法器中形成，如图 5.10 所示。

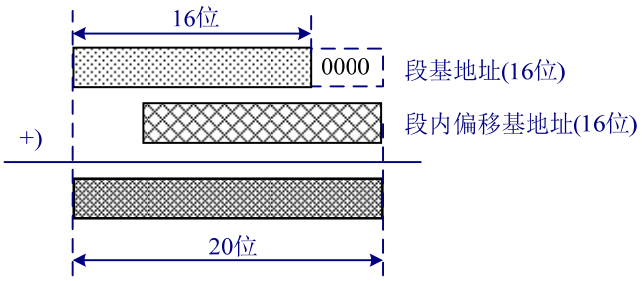


图 5. 10　20 位物理地址的形成

例如：若逻辑地址为 3A00H:12FBH，对应的物理地址是 3B2FBH；若逻辑地址是 8000H:1200H，则对应的物理地址为 81200H。

值得注意的是，一个物理地址所对应的逻辑地址不是惟一的。例如：上例中的物理地址 3B2FBH 的逻辑地址可以是

3A00H:12FBH

3000H:B2FBH

3B00H:02FBH

3．逻辑地址来源

逻辑地址来源如表 5.7 所示。由于访问存储器的操作类型不同，BIU 所使用的逻辑地址来源也不同，取指令时，自动选择 CS 寄存器值作段基址，偏移地址由 IP 来指定，计算出取指令的物理地址。当堆栈操作时，段基址自动选择 SS 寄存器值，偏移地址由 SP 来指定。当进行读／写存储器操作数或访问变量时，则自动选择 DS 或 ES 寄存器值作为段基址(必要时修改为 CS 或 SS)，此时，偏移地址要由指令所给定的寻址方式来决定，可以是指令中包含的直接地址，可以是地址寄存器中的值，也可以是地址寄存器的值加上指令中的偏移量。注意的是当用 BP 作为基地址寻址时，段基址由堆栈寄存器 SS 提供，偏移地址从 BP 中取得。

表 5.7 逻辑地址来源

内存访问类型	默认段寄存器	替换段寄存器	段内偏移地址来源
取指令	CS	无	IP
堆栈操作	SS	无	SP
源字符串	DS	CS、ES、SS	SI
目的字符串	ES	无	DI
BP 用作基址	SS	CS、ES、DS	按寻址方式计算得到的有效地址
一般数据存取	DS	CS、ES、SS	按寻址方式计算得到的有效地址

在字符串寻址时，源操作数放在现行数据段中，段基址由 DS 提供，偏移地址由源变址寄存器 SI 取得，而目标操作数通常放在当前附加段 ES 中，段基址由 ES 寄存器提供，偏移地址从目标变址寄存器 DI 取得。

另外必须注意，在存储器地址的低端和高端，有一些专门用途的存储单元，用于中断和系统复位，用户不能占用，除非专门指定。一般情况下，各段在存储器中的分配由操作系统负责。

5.3.2. 存储器的分体结构

8086 系统中，1MB 的存储空间分成两个存储体：偶地址存储体和奇地址存储体，各为 512KB，示如图 5.11 所示。

图 5. 11　存储器分体结构示意图

当 A0=0 时，选择访问偶地址存储体，偶地址存储体与数据总线低 8 位相连，从低 8 位数据总线读 / 写一个字节。当 $\overline{BHE}$ =0 时，选择访问奇地址存储体，奇地址存储体与数据总线高 8 位相连，由高 8 位数据总线读 / 写一个字节。当 A0=0、$\overline{BHE}$ =0 时，访问两个存储体，读 / 写一个字。A0、$\overline{BHE}$ 功能组合如表 5.8 所示。

表 5.8 A0、$\overline{BHE}$ 功能组合

$\overline{BHE}$	A0	操　作	总线使用情况
0	0	从偶地址开始读 / 写一个字	AD15~AD0
0	1	从奇地址单元读 / 写一个字节	AD15~AD8
1	0	从偶地址单元读 / 写一个字节	AD7~AD0
0	1	从奇地址单元读 / 写一个字	AD15~AD8
1	0		AD7~AD0
1	1	无效	

存储器中存放的信息称为存储单元的内容，例如存储单元 00100H 中的内容为 34H，表示为(00100H)=34H。一个字在存储器中按相邻两个字节存放，存入时以低位字节在低地址，高位字节在高地址的次序存放，字单元的地址以低位地址表示。例：(00100H)=1234H，(00103H)=0152H 在内存中放的位置如图 5.12 所示。从中看出，一个字可以从偶地址开始存放，也可以从奇地址开始存放，但 8086CPU 访问存储器时，都是以字为单位进行的，并从偶地址开始。若读 / 写一个字节，只启动某个存储体，只有相应的 8 位数据在数据总线上有

效，即启动偶地址存储体，低 8 位数据线有效，或启动奇地址存储体，高 8 位数据线有效，另外 8 位数据被忽略了。

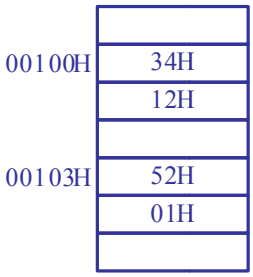


图 5. 12　数据在 8086 存储器中的存放

当 CPU 读 / 写一个字时，若字单元地址从偶地址开始，只需访问一次存储器，低位字节在偶地址单元，高位字节在奇地址单元。若字单元地址从奇地址开始，CPU 要两次访问存储器，第一次取奇地址上数据（偶地址 8 位数据被忽略），第二次取偶地址上数据（奇地址 8 位数据被忽略）。为了加快程序运行速度，编程时注意从存储器偶地址开始存放字数据，这种存放方式也称“对准存放”。

5.3.3. 堆栈的概念

所谓堆栈是在存储器中开辟一个区域，用来存放需要暂时保存的数据。堆栈段是由段定义语句在存储器中定义的一个段，它可以在存储器 1MB 空间内任意浮动，堆栈段容量小于等于 64KB。段基址由堆栈寄存器 SS 指定，栈顶由堆栈指针 SP 指定，根据堆栈构成方式不同，堆栈指针 SP 指向的可以是当前栈顶单元，也可以是栈顶上的一个“空”单元，一般采

用 SP 指向当前栈顶单元。堆栈的地址增长方式一般是向上增长，栈底设在存储器的高地址区，堆栈地址由高向低增长，如图 5.13 所示。

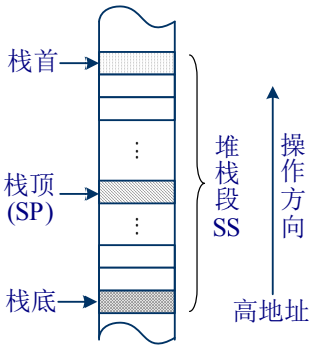


图 5. 13　堆栈的结构

堆栈操作具有如下特点：

① 操作只能在栈顶进行，入栈和出栈都必须是双字节数据(16 位)。进行一次入栈操作，SP 减 2；进行一次出栈操作，SP 加 2；

② 操作遵循“先进后出(FILO)”的原则。最后压入堆栈的数据会最先被弹出。

假设当前 SS=C000H，SP=1000H，先将 FF00H 压入堆栈，再将 3322H 压入堆栈，然后

执行两次出栈操作，指出当前栈顶在存储器中的位置。

解　栈顶的初始位置为 SS:SP= C000H:1000H，如图 5.14 所示。执行一次压栈操作，将 FF00H 压入堆栈，CPU 自动修改指针 SP-2→SP，SP=0FFEH；再执行一次压栈操作，将 3322H 压入堆栈，CPU 自动修改指针 SP-2→SP，SP=0FFCH，此时堆栈中的数据如图 5.14(b)所示。执行两次出栈操作后，SP 恢复为 1000H，如图 5.14 所示。

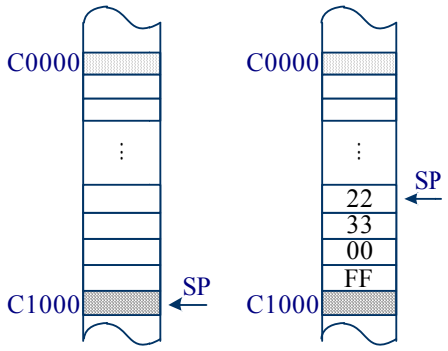


图 5. 14　堆栈操作示意图

5.4. 8086CPU 时序

微处理器是在统一的时钟信号 CLK 控制下，按照一定的时序来工作的。8086 的时钟频率为 5 MHz，故一个时钟周期等于 200 ns。CPU 的时序分为两种：时钟周期和总线周期。8086CPU 与内存或接口间的通信都是通过总线来进行的，如将一个字节写入内存单元，或从内存某单元读一个字节到 CPU，这种通过总线对存储器或 I / O 接口进行一次访问所需的时间叫做一个总线周期，一个总线周期包括多个时钟周期。CPU 每执行一条指令至少要访问一次存储器(取指令)，即至少要进行一次读存储器操作，占用一个读总线周期。

一般地，一条指令的执行需要若干个总线周期才能完成。而一个总线周期又由若干个时钟周期构成。微处理器在运行过程中是按照一个统一的时钟一步步地执行每一个操作的。每个时钟脉冲的持续时间就称为一个时钟周期。8086 CPU 的一个读(或写)总线周期至少包括 4 个时钟周期。显然，时钟周期越短，CPU 执行的速度就越快。典型的总线周期如图 5.15 所示。

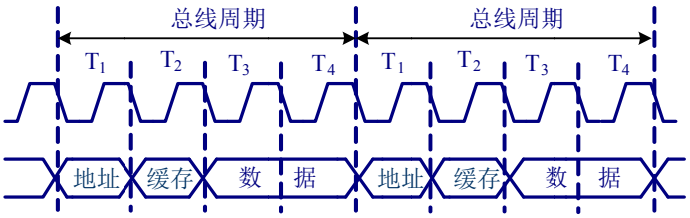


图 5. 15　典型的总线周期

8086 的数据总线和部分地址总线是分时复用的。在一个总线周期内，先利用总线传送地址，将地址锁存后，再利用同一总线传送数据。即在 T1 期间，BIU 部分将要访问的存储单元或输入 / 输出端口地址送上总线，若为读周期，则在第二个时钟周期将总线置为高阻缓

冲状态，以使 CPU 有时间从输出地址方式转换为输人数据方式。之后在 T3 到 T4 期间从总线读入数据到 CPU；若为写周期，则 CPU 就不必转换工作方式，在地址锁存后直接输出数据到总线上。

只有在指令流队列出现 2 个以上空单元时要填补指令流队列或在执行指令的过程中需要申请一个总线周期时，BIU 部分才会进入执行总线周期的状态。在两个总线周期之间，有时可能会出现一些总线上没有信息传送的时钟周期，此时的总线状态称为空闲状态。

5.4.1. 最小模式下的工作时序

8086CPU 在最小模式下的读总线周期和写总线周期时序别如图 5.16 所示。

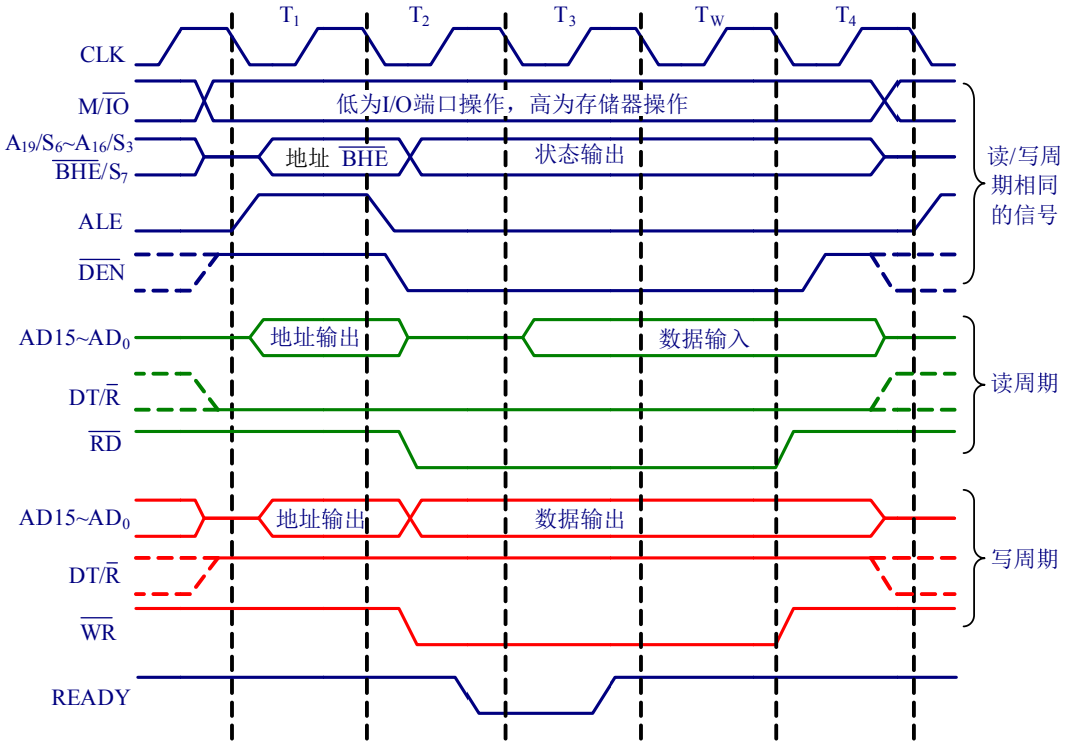


图 5. 16　8086CPU 在最小模式下的读/写总线周期

由前述可知，一个正常的 8086 读(写)总线周期至少需要 4 个时钟周期(Tl～T4)。在 T1 期间，地址/状态复用信号线 A19/S6～A16/S3 和地址/数据复用信号线 ADl5～AD0 分别送出地址 A19～A16 和 A15～A0，与此同时送出地址锁存允许信号 ALE。$\overline{BHE}$ 的状态由 $\overline{BHE}$ /S7 端输出。外部电路在 Tl 后期利用 ALE 的下降沿把地址信号锁存到地址锁存器中，从而在锁存器的输出端得到完整的 20 位地址信号 A19～A0。一旦 ALE 的高电平将地址信号锁存，在此后的时钟周期里，即可利用有关的控制信号如 M/$\overline{IO}$ 、$\overline{RD}$ 、$\overline{WR}$ 等完成对内存或外设的读 / 写操作。读总线周期中，CPU 在 T3 到 T4 期间读入总线上的数据。在写总线周期中，CPU

从 T2 开始把数据送到总线上并维持到 T4。

某些情况下，当内存或接口的速度比较慢，使得在 4 个时钟周期里不能完成读 / 写操作

时，可通过时钟发生器(8284)产生一个低电平信号送到 8086 的 READY 端。8086 在每个总线周期的 T3 开始处都要检查 READY 的状态。若此时 READY 为低电子，则 CPU 不执行 T4，而是在 T3 之后插入一个等待时钟周期 Tw，以等待存储器或 I/O 接口完成读/写操作。在 Tw 的开始时刻，CPU 还要检查 READY 状态，若仍为低电平，则再插入一个 Tw。此过程一直进行到某个 Tw 开始时，READY 已经变为高电平，这时下一个时钟周期就是总线周期的最后一个时钟周期 T4。由此可见，利用 READY 信号，CPU 可以插入若干个 Tw，使总线周期延长，达到可靠地读/写内存和 I/O 接口的目的。

另外还要注意一点，CPU 的读($\overline{RD}$)或写($\overline{WR}$)，是在 T4 开始时刻(或 $\overline{RD}$、$\overline{WR}$ 信号的后沿)进行的，这时数据线上的数据已经到达稳定状态，只有这样，利用 READY 插入 Tw 周期才有意义。

5.4.2. 最大模式下的工作时序

8086 在最大模式下的读总线和写总线周期时序如图 5.17 所示。8086 在最大模式下的工作时序与最小模式下的工作时序有以下几个不同点：

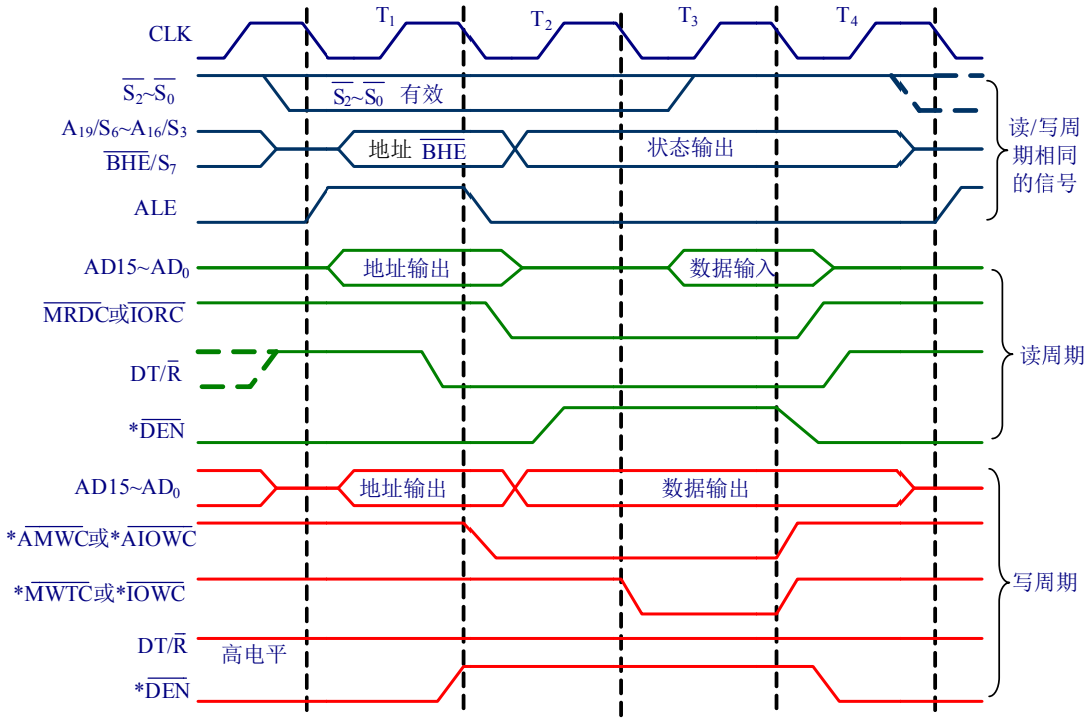


图 5. 17　8086CPU 在最大模式下的读/写总线周期

① 所有地址锁存器和数据驱动器的控制信号在最大模式下都由总线控制器 8288 根据 CPU 输出的 3 个状态位 S0、Sl、S2 的状态产生的；

② 最小模式下的 M/$\overline{IO}$、$\overline{RD}$ 和 $\overline{WR}$ 信号将分别由总线控制器的存储器读命令 $\overline{RD}$ 和 I/O 读命令 $\overline{IORC}$、存储器写命令 $\overline{MWTC}$ 和先行存储器写命令 $\overline{AMWC}$ 及 I/O 写命令 $\overline{IOWC}$、先

行 I/O 写命令 $\overline{\text{AIOWC}}$ 代替；

③ 总线控制器输出的数据允许信号 $\overline{\text{DEN}}$ 与最小模式下 CPU 产生的 $\overline{\text{DEN}}$ 信号极性相反。

5.4.3. 系统的复位和启动操作时序

8086CPU 通过 RESET 引脚上的触发信号来引起 8086 系统复位和启动，复位信号 RESET 至少维持 4 个时钟周期的高电平。当 RESET 信号变成高电平时，8086CPU 结束现行操作，各个内部寄存器复位成初值，如表 5.9 所示。由表可见，代码段奇存器 CS 为 FFFFH，指令指针 IP 清 0，所以 8086 在复位之后重新启动时，从内存的 FFFF0H 处开始执行指令。因此在 FFFF0H 处存放了一条无条件转移指令，转移到系统引导程序的人口处，这样系统启动后就自动进入系统程序。

表 5.9 复位时各内部寄存器的值

标志寄存器 FR	0000H
指令寄存器 IP	0000H
代码段寄存器 CS	FFFFH
数据段寄存器 DS	0000H
堆栈段寄存器 SS	0000H
附加段寄存器 ES	0000H
指令队列	空

在复位时，由于标志寄存器被清 0，所有标志位均为 0，这样从 INTR 引脚进入的可屏蔽中断就被屏蔽了，因此在系统程序中要用开中断指令 STI 来设置中断允许标志。图 5.18 给出了 8086 复位操作时的时序。

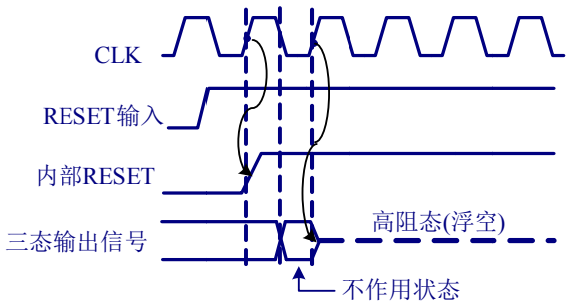


图 5. 18　系统的复位和启动操作时序

在 RESET 信号变成高电平后，经过一个时钟周期，所有的三态输出线被设置成高阻，并一直维持高阻状态（浮空），直到 RESET 信号回到低电平为止。但在高阻状态的前半个时钟周期，三态输出线被置成不作用状态，当时钟信号又变成高电平时，才置成高阻状态。

置成高阻状态的三态输出线包括：AD15～AD0、A19/S6～A16/S3、$\overline{\text{BHE}}$/S7、M/$\overline{\text{IO}}$、DT/$\overline{\text{R}}$、$\overline{\text{DEN}}$、$\overline{\text{WR}}$、$\overline{\text{RD}}$和$\overline{\text{INTA}}$。另外有几条控制线在复位之后处于无效状态，但不浮空，它们是 ALE、HLDA、HOLD、QS0、QS1。

5.5. 8086CPU 寻址方式和指令系统

5.5.1. 8086 的指令格式

控制计算机完成各种指定操作的命令称为指令。一台计算机所能执行的各种指令的集合称为它的指令系统。不同的计算机具有各自不同的指令系统。一个指令系统的功能是否强大与计算机的性能有很大的关系。指令系统的设置与机器的硬件系统结构是密切相关的，指令系统的功能越齐全、通用性越强、指令越丰富，就越需要复杂的硬件结构来支持。

早期的微型计算机因受集成电路技术水平的限制，其硬件结构比较简单，所支持的指令系统一般只有定点数的加减运算、数据传送、转移等几十条最基本的指令。后来，随着集成电路、特别是超大规模集成电路(VLSI)技术的发展，硬件结构变得越来越复杂，功能也不断增强，指令系统也就变得越来越丰富了。在现代计算机的指令系统中，除以上基本指令外，增加了乘除运算、浮点运算、十进制运算、条件转移、子程序调用及字符串处理等指令，另外，为了便于操作系统的实现和优化，还设置了控制系统状态的特权指令、管理多道程序和多处理机系统的专用指令等。

丰富的指令系统大大提高了计算机的性能，但也带来一些负面因素。如因指令系统过于庞大而使设计周期变长、维护困难等。事实上，最经常使用的指令还是数据传送、算术和逻辑运算、转移、子程序调用等几十条最基本的指令。

目前，硬件技术的发展速度已远远超过了软件技术的发展，软件的成本在系统中变得越来越高。为了在不断发展的计算机上能够继承已有的软件以降低软件费用，人们提出了软件“兼容”的概念。这样，在 20 世纪肋年代就出现了系列(Series)计算机。所谓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，如 IBM PC(XT/AT/286/386/486/Pentium)微型计算机系列等。系列机虽然在结构和性能上不断地完善和提高，但其基本结构体系没有变，指令系统向上兼容，即新机种上的指令系统包含了能够在旧机种上运行的全部指令。

指令的格式与机器的字长、存储器的容量及指令的性质都有很大的关系。计算机是通过执行指令来完成各种任务的，微处理器处理的所有对象除指令外都可以认为是数据，所以计算机所要完成的任务就是对各种数据的处理。因此，一条指令至少要包括这样几个信息：运算数据的来源、运算结果的去向及执行的操作。

指令要执行的操作称为操作码，也称为指令码。它说明所执行的操作的性质和功能。操作码有的占用一个字节，有的占用两个字节。一台计算机有几十或几百条指令，每条指令都有一个对应的操作码，计算机通过识别操作码来决定要进行的操作。

参加运算的数据，亦即操作的对象称为操作数。数据的来源及运算结果的去向都统称为操作数的地址，也可简称为操作数。通常将存放结果的操作数称为目标操作数，相应的另一个操作数就称为源操作数。

除操作码和操作数外，在遇到转移、子程序调用等非顺序执行的情况时，指令中还应给出下一条要执行的指令的地址(在程序顺序执行时，下条要执行的指令的地址由指令指针 IP 指出)。

总之，一条指令主要包括了两种信息，即操作码和操作数(或称地址码)。操作码(OP Operation Code)部分描述操作的性质(如加、减、乘、除等)，操作数(Operation Data)描述了操作的对象，它可以是直接参加运算的数据，也可以是数据的地址。如何确定操作数的地址，就要涉及操作数的寻址方式。

指令的长度与操作码和操作数的多少及其类型都有关系。操作数越多，其指令的长度就越长。一条 8086 指令的长度一般在 1~7 个字节之间。第一个字节通常为指令的操作码，如图 5.19 所示。

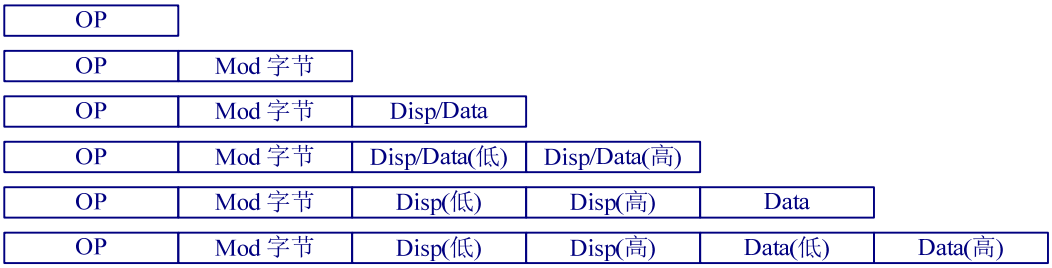


图 5. 19　不同字长的指令格

1．零操作数指令

这种指令格式中只有操作码 OP 而没有操作数。这种指令一般有两种：一是不需要任何操作数。如空操作指令 NOP、暂停指令 HLT 等；二是操作数是隐含的，即指令有默认的操作数。如字符串操作指令 MOVSB，该指令表示将源地址中的字节数传送到目标地址中去。

2．单操作数指令

这里操作数有两种含义：①该操作数既表示运算数据的地址，也表示存放运算结果的地址。如加 1 指令：INC AX，该指令执行的结果是将累加器 AX 中的内容加 1 然后再送回 AX；②指令中的操作数是源操作数，而目标操作数被隐含了，即指令实际上是双操作数指令，默认了另一个操作数，运算的结果也默认存放在隐含的操作数地址中。如乘法指令：MUL BL，该指令的执行是将累加器 AL 的内容与 BL 内容相乘，结果送 AX。

3．双操作数指令

指令中的 Data 是源操作数，它本身就是参加运算的一个数据，或是其中一个操作数的地址；disp 是目标操作数，它有两个含义，表示另一个操作数的地址及存放运算结果的目标地址。这是最常见的指令格式，微处理器的基本指令系统中多数指令都是双操作数指令。例如：加法指令：ADD AX，BX。指令中的源操作数是 BX，表示参加运算的操作数之一是 BX 的内容，AX 为目标操作数，它既表示参加运算的另一数据是 AX 的内容，同时也表示运算的结果存放在 AX 中。该指令执行的结果是将 AX 的内容加上 BX 的内容，结果送 AX 中。

微型计算机中的指令都是以上三种格式。在大型机或某些功能较强的中、小型机中，常置有一些功能很强、用于处理成批数据的指令，如字符串处理、向量及矩阵运算等。在这些指令中，有时需要用多个操作数来描述数据存放的首地址、数据长度、下标等信息。这一类

的指令格式就不再限于以上三种，而可以是三操作数以至多操作数的指令格式。

在计算机中，无论是指令码还是数据都是以二进制码的形式存放在存储器中的，仅从表面上看不出二者有什么区别。但指令的地址是由指令指针 IP 确定，数据的地址则是由指令中给出，CPU 在访问存储器时是绝对不会将指令和数据混淆的。

8086 指令系统中的指令均为以上三种格式。其操作数既可以是数据，也可以是参加运算的数据的地址。根据它们的性质，将操作数分为这样三类：立即数操作数、寄存器操作数和存储器操作数。

所谓立即数是指具有固定数值的操作数，即常数。它不会由于指令的执行而发生变化。它可以是字节(8 位)或字(16 位)，当它们分别代表无符号数和有符号数时，其各自的取值范围见表 5.10。

表 5.10 立即数操作数的取值范围

	8 位数	16 位数
无符号数	00H~0FFH(0~255)	0000H~0FFFFH (0~65535)
带符号数	80H~7FH(-128~+127)	8000H~7FFFH (-32768~+32767)

如果一个立即数的取值超出了规定的范围，就会发生错误。在指令中，立即数操作数表示参加运算的数据而不是数据的地址，所以只能用作源操作数，而不能作为目标操作数。

寄存器操作数是 8086 的 8 个通用寄存器或段寄存器，表示数据的地址或运算结果的地址。它既可以用做源操作数，也可以用做目标操作数。

通用寄存器主要用于存放操作的数据。通用寄存器中的 AX、BX、CX、DX 既可以作为 4 个 16 位寄存器，用来存放字操作数，也可以当作 8 个 8 位寄存器(AH、AL、BH、BL、CH、CL、DH、DL)，用来存放字节操作数。SI、DI、BP、SP 只能存放字操作数。

段寄存器用来存放当前操作数据的段基地址。在与通用寄存器或存储器传送数据时，段寄存器可作为源操作数或目标操作数(但代码段寄存器 CS 一般不作为目标操作数)。此外，8086 不允许用一条指令将立即数传送到段寄存器。如果需要这样做，可用某个通用寄存器作为中间桥梁，用两条传送指令实现。

仅有个别指令将标志寄存器 FLAGS 作为指令的操作数。

存储器操作数表示当前操作数据的地址或运算结果的地址。故在指令中既可作为源操作数，也可以作为目标操作数。操作的数据可以是字节、字或双字，分别存放在 1 个、2 个或 4 个存储单元中。但在 8086 中，大多数指令不允许源操作数和目标操作数同时为存储器操作数，也就是说，不允许从存储器到存储器的操作。若有这样的需要，可以先将其中一个存储器的内容传送到某个通用寄存器中，然后再把这个寄存器与另一个存储器的内容作为操作数执行希望的操作。

由于能够惟一标识一个存储器单元的是它的物理地址，而物理地址是由段基地址和偏移地址两部分构成，因此，要寻找一个存储在存储器中的操作数，必须首先确定数据所在的段，即确定有关的段寄存器。一般情况下，若指令中没有指明所涉及的段寄存器，则 CPU 就采用默认的段寄存器来确定数据所在的段。各种存储器操作所约定的默认段寄存器、段超越(即

显式地指明段寄存器)所允许的段寄存器以及指令的有效地址所在的寄存器见表 5.11。

表 5.11 段寄存器使用的一些基本约定

存储器操作类型	默认的段寄存器	允许超越的段寄存器	段内偏移地址来源
取指令	CS	无	IP
堆栈操作	SS	无	SP
通用数据读 / 写	DS	CS，ES，SS	按寻址方式取得
串操作源串地址	DS	CS，ES，SS	SI
串操作目标串地址	ES	无	DI
BP 作为基址寄存器	SS	CS，DS，ES	按寻址方式取得

5.5.2. 8086 的指令的执行

1．指令字长与机器字长

机器字长是指计算机能够直接处理的二进制数的位数，是计算机的一个重要技术指标，它决定了计算机的运算精度，字长越长，运算的精度就越高。因主存储器都是按字节编址的，每个地址单元存放 8 位二进制码，为尽可能充分利用存储空间并便于数据处理，机器的字长都设计成字节长度的 1、2、4、或 8 倍，即 8 位、16 位、32 位或者 64 位，对微型计算机来讲，目前其字长一般都是 32 位，即计算机能够直接处理 32 位二进制数。

指令的字长包括操作码的长度、操作数地址的长度及操作数的个数。在微型计算机中，操作码的长度是不固定的(操作码采用可变格式的编码格式)，操作数的格式也不同，所以指令的长度也是不固定的。为了充分利用存储空间，指令长度一般为字节的整数倍。例如：8086CPU 的指令长度为 1~7 个字节。

8086CPU 的指令系统采用变字长的指令格式。指令中包括操作码、寻址方式以及操作数三部分信息，其指令编码的一般形式如图 5.20 所示。指令第一个字节是操作码，说明操作的性质；第二个字节规定操作数的寻址方式，包括 Mod(方式)、Reg(寄存器)/操作码和 R/M(寄存器 / 存储器)三个域，用于操作数的类型及寻找它的方法。其中：

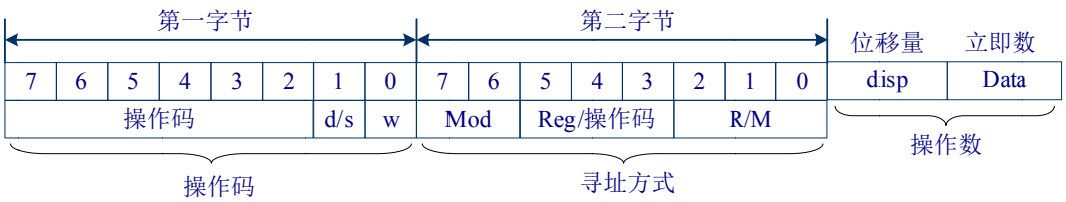


图 5. 20　8086 指令编码的一般格式

① Reg 域用来规定一个操作数为寄存器操作数，至于它是源操作数还是目标操作数则由操作码确定；

② Mod 域用来确定另一个操作数是寄存器操作数还是存储器操作数，并在必要时规定后面的位移量长度；

③ 当 Mod 域确定另一个操作数是寄存器操作数时，它负责给出寄存器号。若是存储器操作数则指出如何求得存储单元的有效地址。

最后的字节是操作数，它的长度由 Mod 域确定。

字节中各符号的含义如下：

d：进行寄存器寻址的标志，部分双操作数指令有效。d=0，源操作数为寄存器；d=1，目标操作数为寄存器。

w：字标志操作位，部分指令有效。w=0，表示当前指令进行字节操作；w=1，表示当前指令进行字操作。

s：符号扩展位，部分指令有效。s=0，对操作数不进行符号扩展；当 s=1，w=1 时，对 8 位立即数操作数进行符号扩展，得到 16 位操作数。

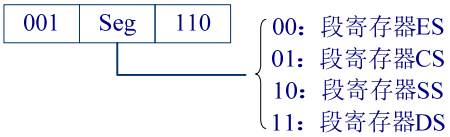
Mod：方式字段。用于指示操作数是在存储器中还是在寄存器中。

R/M：与 Mod 字段组合，说明操作数所在的寄存器或存储器单元地址的计算方法，即寻址方式，见表 5.12 所示。

表 5.12 Mod 与 R/M 字段组合编码及有关地址计算

Mod / R/M	00	01	10	11	
				w=0	w=1
000	DS:[BX+SI]	DS:[BX+SI+disp8]	DS:[BX+SI+disp16]	AL	AX
001	DS:[BX+DI]	DS:[BX+DI+disp8]	DS:[BX+DI+disp16]	CL	CX
010	SS:[BP+SI]	SS:[BP+SI+disp8]	SS:[BP+SI+disp16]	DL	DX
011	SS:[BP+DI]	SS:[BP+DI+disp8]	SS:[BP+DI+disp16]	BL	BX
100	DS:[SI]	DS:[SI+disp8]	DS:[SI+disp16]	AH	SP
101	DS:[DI]	DS:[DI+disp8]	DS:[DI+disp16]	CH	BP
110	DS:disp16	DS:[BP+disp8]	DS:[BP+disp16]	DH	SI
111	DS:[BX]	DS:[BX+disp8]	DS:[BX+disp16]	BH	DI

表 5.12 中的段寄存器是指无段超越的情况下所使用的隐含的段寄存器。如果指令中指定段超越前缀，则在机器语言中使用放在指令之前的一个字节来表示，其格式为：



Reg/操作码：第二字节的 D3~D5 位既可以是寄存器字段，也可以是操作码。如果是操作码，用于单操作数指令（或含两个操作数，但有一个操作数隐含在操作码中）；如果是寄存器字段，用于双操作数指令，规定一个寄存器操作数，如表 5.13 所示。

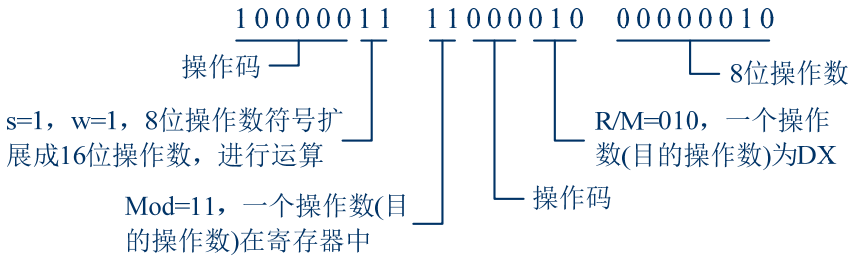
表 5.13 Reg 字段编码表

Reg	w=0（字节操作）	w=1（字操作）
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

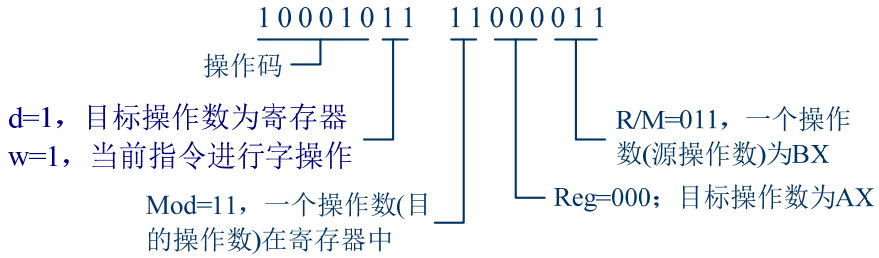
下面举例说明 8086 指令的编码格式。

指令 ADD DX, 02H 的编码格式。

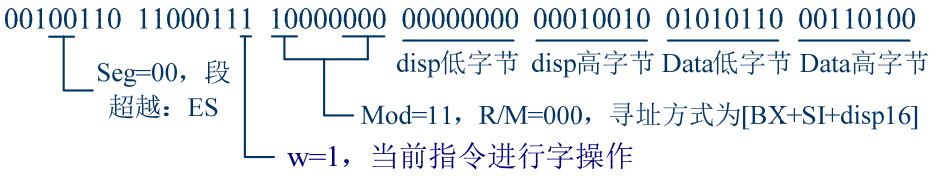
11010101　00001010



指令 MOV AX, BX 的编码格式。



指令 MOV ES:[BX+SI+1200H], 3456H 的编码格式。



指令的长度可以小于或等于机器的字长，也可以大于机器字长。前者称为短格式指令，所占的存储空间比较小，执行速度快，一般都将最常用的指令(如数据传送、算术运算等)设计成短格式。对于操作复杂的指令，因所需的信息量大，则设计为长格式。

2．指令的执行时间

了解指令的执行时间有时是很重要的。例如在用软件实现定时或延时时，需要估算出一段程序的运行时间。另外，在某些实时控制要求较严或对程序运行时间要求较高的场合，除需认真研究程序的算法外，对选择什么样的指令及采用什么样的寻址方式也是很重要的。因为不同的指令在执行时间上有很大的差别，而不同的寻址方式其计算偏移地址所需时间也不同。由于指令的种类很多，要详细讨论各种指令的执行时间比较困难，这里仅以 8088CPU 指令系统为例，对指令的执行时间作一般的讨论。

一条指令的执行时间应包括取指令、取操作数、执行指令及传送结果几个部分，单位用时钟周期数表示。

不同指令的执行时间有较大的差别，表 5.14 列出了部分常用指令的执行时间及其访问存储器的次数。

存取操作数的时间与采用的寻址方式有关。寄存器操作数占用的时间最短，而对存储器操作数则还需考虑计算偏移地址所花的时间。表 5.15 为不同寻址方式下，计算偏移地址所需要的时间。

表 5.15 8088CPU 常用指令执行时间

指　　令		所需时钟周期数	访问内存次数
MOV	累加器到内存	10(14)	1
	内存到累加器	10(14)	1
	寄存器到寄存器	2	0

	内存到寄存器	8(12)+EA	1
	寄存器到内存	9(13)+EA	1
MOV	立即数到寄存器	4	0
	立即数到内存	10(14)+EA	1
	寄存器到段寄存器	2	0
	内存到段寄存器	8(12)+EA	1
	段寄存器到寄存器	2	0
	段寄存器到内存	9(13)+EA	1
ADD 或 SUB	寄存器到寄存器	3	0
	内存到寄存器	9(13)+EA	1
	寄存器到内存	16(24)+EA	2
	立即数到寄存器	4	0
	立即数到内存	17(25)+EA	2
MUL	累加器乘 8 位寄存器	70~77	0
	累加器乘 16 位寄存器	118~133	0
	累加器和内存字节乘	(76~83)+EA	1
	累加器和内存字乘	[124(128)~139(143)]+EA	1
IMUL	累加器乘 8 位寄存器	80~98	0
	累加器乘 16 位寄存器	128~154	0
	累加器和内存字节乘	(86~104)+EA	1
	累加器和内存字乘	[134(138)~160(164)]+EA	1
DIV	除数在 8 位寄存器中	80~90	0
	除数在 16 位寄存器中	144~162	0
	除数为 8 位内存数	(86~96)+EA	1
	除数为 16 位内存数	[150(154)~168(172))+EA	1
IDIV	除数在 8 位寄存器中	101~112	0
	除数在 16 位寄存器中	165~184	0
	除数为 8 位内存数	(107~118)+EA	1
	除数为 16 位内存数	[171(175)~190(194)]+EA	1

循环和移位	在寄存器中移 1 位	2	0
	在寄存器中移若干位	8+4×位数	0
	内存数据移 1 位	15(23)+EA	2
	内存数据移若干位	20(28)+EA+4×位数	2
JMP	段内／段间直接转移	15	0
	段内间接转移	8(12)+EA	1
	段间间接转移	24(32)+EA	2
条件转移	JCXZ	6(不转移)	0
		18(转移)	0
	其他条件转移指令	4(不转移)	0
		16(转移)	0

注：①表中 EA 表示偏移地址。小括号内的数为 8088 进行字操作的时钟数。因为 8088 的数据线只有 8 位，每个总线周期只能传送一个字节，所以对字操作要再加上 4 个时钟周期。

②对条件转移指令，若条件满足，执行的时间比较长。因为要产生转移，就要包括取下一条指令所需的时间。若条件不满足，执行时间就较短。因为此时不产生转移，而是执行下一条指令。

表 5.15 计算偏移地址 EA 所需时间

寻址方式		计算 EA 所需时钟数
直接寻址		6
寄存器间接寻址		5
寄存器相对寻址		9
基址、变址寻址	[BX+S1)，[BX+D1]	7
	[BP+S1)，[BP+D1)	8
基址、变址加相对寻址	[BX+SI+位移量]，[BP+DI+位移量]	11
	[BX+DI+位移量]，[BP+SI+位移量]	12

注：①若有段超越，则需再加上两个时钟周期。

②寻址方式的介绍参见 5.2 节。

从上面的表中可以看到，对同一种指令，如果寻址方式不同，其指令执行时间可能相差很大。在以上讨论的三种类型的操作数中，寄存器操作数的指令执行速度最快，立即数操作数次之，存储器操作数指令的执行速度最慢。这是由于寄存器位于 CPU 的内部，执行寄存

器操作数指令时，8086 的执行单元(EU)可以简捷地从 CPU 内部的寄存器中取得操作数，不需要访问内存，因此执行速度很快。立即数操作数作为指令的一部分，在取指时被 8086 总线接口单元(BIU)取出后存放在 BIU 的指令队列中，执行指令时也不需要访问内存，因而执行速度也比较快。而存储器操作数放在某些内存单元中，为了取得操作数，首先要由总线接口单元计算出其所在单元的 20 位物理地址，然后再执行存储器的读 / 写操作。所以相对前述两种操作数来说，指令的执行速度最慢。

以通用数据传送指令(MOV)为例，若 CPU 的时钟频率为 5 MHz，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=2×0.2=0.4μs

立即数传送到寄存器的指令执行时间为

t=4×0.2=0.8μs

而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+EA)×0.2=(8+8)×0.2=3.2μs

5.5.3. 8086 的寻址方式

所谓寻址方式，是指寻找本条指令中数据的地址或是指在转移类指令中确定转移的目标地址的方法。前者称为寻找操作数的寻址方式，后者称为寻找指令地址的寻址方式。寻址方式是指令系统中一个很重要的内容。它与计算机的硬件结构密切相关，对指令的格式和功能都有很大的影响。在汇编语言的程序设计及高级语言的编译程序设计中，都需要对计算机的寻址方式非常了解。

1. 寻找操作数的寻址方式

在程序的执行过程中，操作数可能存放在寄存器中，也可能存放在主存储器或 I / O 端口中，或者由指令直接给出(此时的操作数事实上也在存储器中)。对于指令中直接给出的立即数操作数或寄存器操作数，其寻址方法是很简单的，比较复杂的是寻找存放在存储器中的操作数。下面讨论 8086CPU 的几种基本寻址方法。

(1) 立即寻址

立即寻址(1mmediateAddressing)是指指令中的源操作数是参加操作的一个立即数，由指令直接给出。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。

这种寻址方式减少了 CPU 访问存储器的次数，使指令的执行速度提高。但由于立即数是指令的一部分，在程序运行中不能改动，而多数情况下指令处理的数据是在不断变化的，如上条指令执行的结果可能是下条指令处理的数据。所以立即寻址方式使用的范围很窄，主要用于给寄存器或存储单元赋初值。

指令中的立即数可以是 8 位或 16 位的整数。若为 16 位，则存放时低 8 位存放在低地址单元，高 8 位存放在高地址单元。

指令"MOVAX，2233H"表示将 16 位的立即数 2233H 送入累加器 AX。指令执行后，AH=22H，AL=33H。

这是一条三字节指令，图 5.21 给出了指令执行情况示意图。

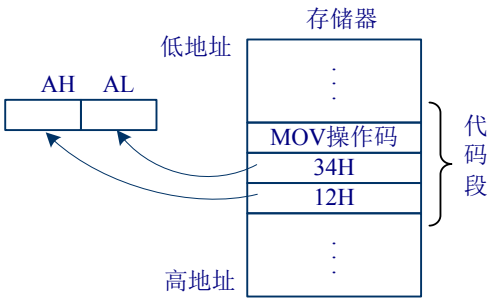


图 5. 21　立即寻址操作示意图

(2) 直接寻址

直接寻址(DirectAddressing)指令是在指令中直接给出操作数的 16 位偏移地址，该地址与指令的操作码一起存放在内存的代码段，也是低 8 位在低地址，高 8 位在高地址。但是，若操作数本身在指令中无特殊声明，默认存放在内存的数据段中。

指令“MOV AX，[1234H]”将数据段中偏移地址为 1234H 和 1235H 两单元的内容送到 AX 中。

这时，如果(DS)=4000H，则所寻找的操作数的物理地址为

40000H+1234H=41234H

指令的执行情况如图 5.22 所示。

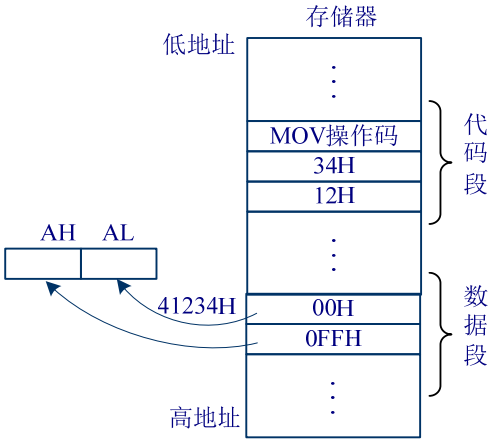


图 5. 22　直接寻址方式

要注意区别直接寻址指令与前面介绍的立即寻址指令二者的不同。直接寻址指令中的数值是操作数的 16 位偏移地址，而不是操作数本身。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。

在上例中，指令的执行不是将立即数 1234H 送到累加器 AX，而是将偏移地址为 1234H 的内存单元中的内容送 AX。若操作数不是存放在 DS 段，则在指令中要用段超越符号加以声明。

指令“MOV BX，ES :[1200H]”将附加段中偏移地址为 1200H 和 1201H 两单元的内容送

到 BX 寄存器中。

在汇编语言中，有时也用一个符号来代替数值以表示操作数的偏移地址，通常把这个符号称为符号地址。上例中，若用 BUFFER 代替偏移地址 1200H，则指令可写成 MOV BX，ES:[BUFFER]，这两者是等效的，但 BUFFER 必须在程序的起始处予以声明。

(3) 寄存器寻址

寄存器寻址(Resister Addressing)中指令的操作数为 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。

指令“MOV SS，AX”将 AX 的内容送到寄存器 SS 中，其执行情况如图 5.23 所示。

若指令执行前(AX)=0FD3H，SS=4455H，则指令执行后(SS)=0FD3H，而(AX)中的内容保持不变。

图 5. 23　寄存器寻址示意图

采用寄存器寻址时，虽然指令的操作码在代码段中，但操作数在 CPU 的寄存器中，指令在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。

(4) 寄存器间接寻址

寄存器间接寻址(Register Indirect Addressing)与寄存器寻址方式不同，指令中指定的寄存器的内容不是操作数，而是操作数的偏移地址。也就是说操作数的偏移地址放在寄存器中，操作数本身则在存储器中。存放存储器地址的寄存器有时也称为地址指针。寄存器间接寻址方式可用的寄存器只允许是 SI、DI、BX 和 BP 4 个，它们可简称为间址寄存器。选择不同的间址寄存器，涉及的段寄存器将有所不同。在默认情况下，选择 SI、DI、BX 为间址寄存器时，操作数在数据段，段基地址由 DS 决定；选择 BP 为间址寄存器，则操作数在堆栈段，段基地址由 SS 决定。但无论选择哪一个间址寄存器，都允许段超越，即可在指令中用段超越前缀指明当前操作数在哪一个段。

因为间址寄存器中存放的是操作数地址，所以根据规定，用作间址的寄存器必须加上方括号，以避免与寄存器寻址指令混淆。

指令“MOV AX，[BX]”的寻址过程如图 5.24 所示。

已知(DS)=4000H，(BX)=1200H。因为指令中没有特别的声明(指定段超越)，所以操作数默认在数据段。可计算出操作数的物理地址为 40000H+1200H=41200H，执行的结果为

(AX)=0FE00H

若操作数是在附加段，则本例中的指令应表示成以下形式：

MOV AX，ES：[BX]

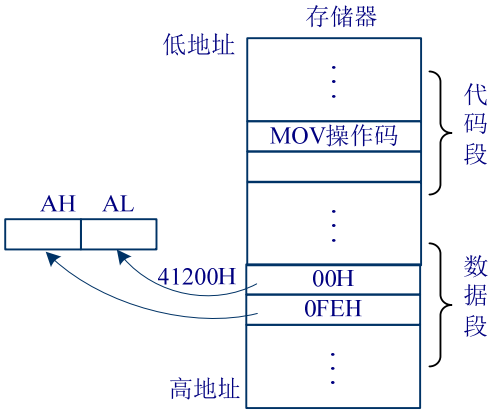


图 5. 24　寄存器间接寻址执行示意图

在指令"MOVBX，[BP]”中，由于用 BP 作间址寄存器，故操作数默认在堆栈段(SS 段)中。若已知 SS=8000H，BP=0200H，则操作数的物理地址为

80000H+0200H=80200H

指令执行结果为：BL=[80200H]中的内容，BH=[80201]中的内容。

由于 4 个间址寄存器包括两个基址寄存器 BX、BP 以及两个变址寄存器 SI、DI，因此可将寄存器间接寻址方式分为基址寻址和变址寻址。基址寻址方式使用 BX、BP 作为间址寄存器，而变址寻址方式使用 SI 和 DI 作间址寄存器。

(5) 寄存器相对寻址

采用这种寻址方式时，存放于主存中的操作数的偏移地址等于间址寄存器的内容加上指令中给出的一个 8 位或 16 位的地址位移量，位移量紧随操作码一起存放在代码段中。操作数默认在哪一个段，则仍由所使用的间址寄存器决定(使用 BP 则默认在堆栈段，其他的则默认在数据段)。位移量也可看作相对值，故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。

指令“MOV AL，[BX+05]”的寻址过程示例。

设：(DS)=4000H，(BX)=1200H，则

物理地址=40000H+1200H+05H=41205H

指令的执行情况如图 5.25 所示。执行结果为

(AL)=00H

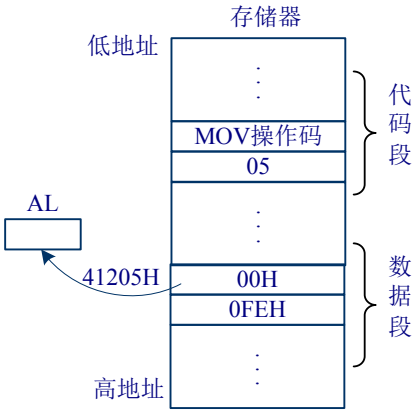


图 5. 25　寄存器相对寻址示意图

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，元素的下标值放在间址寄存器中(反过来也可以)，即可存取表格中的任意一个元素。

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

MOV AH，DATA[SI]

MOV AH，[SI]DATA

MOV AH，DATA+[SI]

MOV AH，[SI]+DATA

MOV AH，[DATA+SI]

MOV AH，[SI+DATA]

(6) 基址-变址寻址

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在主存中的偏移地址。数据的段基地址在默认情况下则仍由指令中使用的基址寄存器决定，即若使用 BX 作基址寄存器，则数据默认在数据段；若使用 BP 作基址寄存器，则数据默认在堆栈段。指令允许段超越。

指令 MOV AX, [BP+SI]的寻址过程示例。

已知 SS=2000H，BP=1000H，SI=0006H，则物理地址=20000H+1000H+0006H=21006H。指令执行后：

(AH)=0FEH，(AL)=00H

指令执行情况如图 5.26 所示。

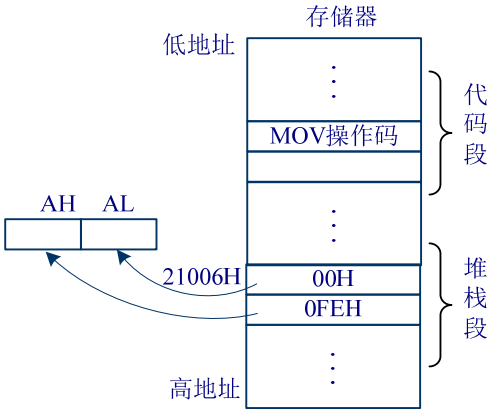


图 5. 26　基址-变址寻址

使用基址—变址寻址方式时，8086CPU 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

(7) 基址-变址-相对寻址

这种寻址方式事实上是上一种方式的扩充。指令中指定了一个基址寄存器和一个变址寄存器，同时还给出一个 8 位或 16 位的位移量，将三者相加就得到操作数在主存中的偏移地址。至于默认的段寄存器，仍由所用的基址寄存器决定。指令同样允许段超越。

指令"MOV AX，[DI+BX+06H]”的寻址过程示例。

已知 DS=2000H，BX=1000H，DI=2000H，而偏移地址为

(BX)+(DI)+06H=1000H+2000H+06H=3006H。则物理地址=20000H+3006H=23006H。指令执行后：

(AH)=0FEH，(AL)=00H

指令的执行情况如图 5.27 所示。

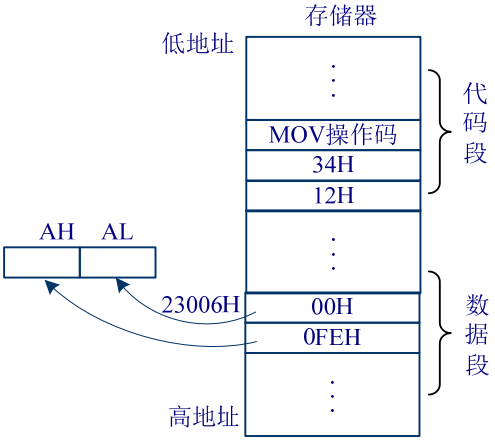


图 5. 27　基址-变址-相对寻址示意图

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)，而变址寄存器和位移量分别存放行和列的值，则利用基址-变址-相对寻址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[S1][BX]

MOV AX，[BX+DATA][S1]

MOV AX，[BX+SI+DATA]

MOV AX，[BX]DATA[SI]

MOV AX，[BX+SI]DATA

同样地，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]

MOV AX，[BX][BP]DATA

(8) 隐含寻址

在有些指令的操作码中，不仅包含了操作的性质，还隐含了部分操作数的地址。如乘法指令 MUL，在这条指令中只需指明乘数的地址，而被乘数以及乘积的地址是隐含且固定的。这种将一个操作数隐含在指令码中的寻址方式就称为隐含寻址。

指令“MUL BL”的功能是把 AL 中的内容与 BL 中的内容相乘，乘积送到 AX 寄存器。即(AL)×(BL)→AX。这条指令隐含了被乘数 AL 及存放结果的累加器 AX。

2. 寻找转移地址的寻址方式

在一般情况下，当 BIU 取走一条指令后，指令指针 IP 会自动加 1 指向代码段中下一条要执行指令的地址，使程序按照预先设定好的次序，由低地址到高地址顺序执行。但有时需要改变程序的这种执行顺序，而转移到一个新的地址再顺序执行。因 BIU 是严格按照 CS 和 IP 所给出的地址去取指令的，所以，只要修改 CS 和 IP 的内容就能够实现程序转移到新地址继续执行的目的。寻找转移地址的寻址方法就是找出转移的目标地址，也可以说是下一条要执行指令的地址，而不再是操作数地址。

程序的转移可以是在当前代码段内，这种转移称为段内转移；程序也可以转移到另一个代码段，这称为段间转移。因此转移地址的寻址相应地有段内寻址和段间寻址两种方式。

(1) 段内寻址方式

1) 段内直接寻址

段内直接寻址也称为相对寻址。转移的地址是当前 IP 的内容加上指令给定的 8 位或 16 位位移量，形成新的 IP，并使 CS 的内容保持不变。位移量可以为正，也可以为负。如果位移量是 8 位，称为段内直接短转移，转移范围为-128~+127；若位移量为 16 位，称为段内直接近转移，转移范围为-32 768~+32 767。

指令中常用字符标号指定位移量，即由汇编程序计算程序中的标号与转移指令之间的距离，也就是位移量。

段内直接寻址方式适合于条件转移或无条件转移类指令，但条件转移指令只能是段内短转移。

2) 段内间接寻址

此时程序转移的地址存放在寄存器或存储器的连续两单元中。执行时用寄存器或存储单元的内容取代当前 IP 的值。存储单元在数据段，其偏移地址根据转移指令指定的寻找操作数的寻址方式得出。

(2) 段间寻址方式

使用段间寻址方式是表示程序转移的目标地址不在当前代码段内。亦即不仅要改变指令指针 IP 的内容，代码段寄存器 CS 的内容也要修改。段间寻址同样可分为直接寻址和间接寻址两种方式。

1) 段间直接寻址

这种寻址方式是直接用指令中给出的 16 位的段地址以及 16 位的偏移地址来取代当前 CS 和 IP 的内容。

2) 段间间接寻址

此时转移的目标地址存放在存储器的连续 4 个单元中，高地址单元存放新的 CS 值，低地址单元存放新的 IP 值。这 4 个单元在数据段中，它们的偏移地址也同样根据转移指令指定的寻址方式得出。

5.5.4. 8086 的指令系统

一台计算机的指令系统通常有几十到几百条指令，它们决定了计算机的基本功能。8086CPU 共有 96 条指令，其常用指令及助记符见表 5.16。按照它们的功能大致可分为以下六大类：

① 数据传送；

② 算术运算；

③ 逻辑运算和移位；

④ 串操作；

⑤ 程序控制；

⑥ 处理器控制。

下面分别说明各类指令的格式、执行的操作及其功能。

表 5.16 8086 CPU 常用指令一览表

指令类型		助记符
数据传送	数据传送	MOV，PUSH，POP，XCHG，XLAT，CBW，CWD
	输入/输出	IN，OUT
	地址传送	LEA，LDS，LES
	标志传送	LAFH，SAFH，PUSHF，POPF
算术运算	加法	ADD，ADC，INC
	减法	SUB，SBB，DEC，NEG，CMP
	乘法	MUL，IMUL
	除法	DIV，IDIV
	十进制调整	DAA，AAA，DAS，AAS，AAM，AAD
逻辑运算和移位		AND，OR，NOT，XOR，TEST，SHL，SAL，SHR，SAR，ROL，ROR，RCL，RCR
串操作		MOVS，CMPS，SCAS，LODS，STOS
控制转移		JMP，CALL，RET，LOOP，LOOPE，LOOPNE，INT，INTO，IRET，各类条件转移指令
处理器控制		CLC，STC，CMC，CLD，STD，CLI，STI，HLT，WAIT，ESC，LOCK，NOP

一、数据传送指令

数据传送指令是将数据、地址或立即数传送到寄存器或存储单元中。它又可分为通用数据传送指令、输入输出指令、目的地址传送指令和标志传送指令等四组。

数据传送指令对状态标志位不发生影响，除(SAHF 和 POPF)例外，下面分别进行讨论。

1. 通用数据传送指令

1) 数据传送指令

指令格式及操作：

MOV dst，src ；(dst)←(src)

指令中，dst 为目的操作数，src 为源操作数，指令实现的操作是将源操作数传送到目的操作数地址。这种传送实际上是进行数据的“复制”，将源操作数复制到目标操作数地址中，源操作数保持不变。

双操作数的书写方法一般总是将目标操作数写在前面，源操作数写在后面，两者之间用逗号隔开。在 MOV 指令中，源操作数可以是寄存器、存储器、段寄存器和立即数；目标

操作数可以是寄存器(不能为 IP)、存储器、段寄存器(不能为 CS)。除了源操作数和目标操作数不能同时为存储器、段寄存器、立即数送段寄存器外，可任意搭配。数据传送的方向如图 3.5 所示：数据传送类指令是程序中使用最为频繁的一类指令，因为无论什么样的程序，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多次传送。数据传送时，数据从源地址传送到目标地址，且源地址中的数据保持不变，即相当于数据拷贝的操作。

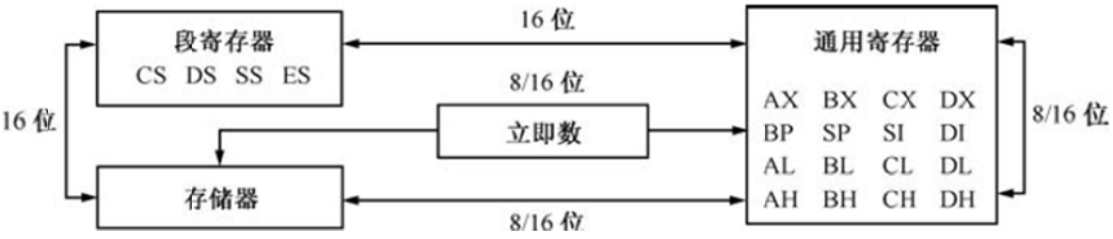


图 5. 28　MOV 指令的数据传送方向

2) 堆栈操作指令

堆栈操作指令是用来完成压入和弹出堆栈操作的。

(1) 压入堆栈指令。

指令格式及操作：

PUSH src ； SP←SP－2，(SP)＋1：(SP)←(src)

指令完成的操作是“先减后压”，先将指针 SP 减 2，然后再将操作数 src 压入由 SP 指出的栈顶中，指令中的操作数可以是通用寄存器、段寄存器、存储器，但不能是立即数。例如：　PUSH AX ；　SP←SP－2，(SP)＋1←(AH)，(SP)←(AL)

PUSH CS

PUSH [SI]

(2) 弹出堆栈指令。

指令格式及操作：

POP dst ； dst ← ((SP)＋1，(SP))，SP←(SP)＋2

指令完成的操作是“先出后移”，即先将堆栈指针 SP 所指示的栈顶存储单元的值弹出到操作数 dst 中，然后再将堆栈指针 SP 加 2。指令中的操作数可以是通用寄存器、存储器、段寄存器(但不能是代码段寄存器 CS)，同样，不能是立即数。例如：

POP BX ； BH←((SP)＋1)，BL＋((SP))，SP←(SP)＋2

POP ES

POP MEM[DI]

应该注意，堆栈指令中操作数一定是字操作数(16 位)。

3) 数据交换指令

指令格式及操作：

XCHG oprl，opr2 ； oprl←→ (opr2)

这是一条交换指令，它的操作是使源操作数与目标操作数进行交换，即不仅将源操作数传送到目标操作数，而且同时将目标操作数传送到源操作数。交换指令的源操作数与目标操作数均可以是通用寄存器、存储器，但不能同时为存储器。

4) 字节查表转换指令

指令格式及操作：

XLAT src-table

XLAT 指令可以根据表中元素序号，查出表中相应元素的内容。为了实现查表转换，预先应将表的首地址，即表头地址传送到 BX 寄存器，元素的序号即位移量送 AL，表中第一个元素的序号为 0，然后依次是 1，2，3，…。执行 XLAT 指令后，表中指令序号的元素存于 AL。由于借助了 AL 寄存器进行，所以被寻址的表的最大长度为 255 个字节。利用 XLAT 指令实现不同数制或编码系统之间的转换十分方便。

内存的数据段有一个 0～9 的 ASCII 码表，其首地址为 ASCII，现在要求查出元素“5”的 ASCII 码，实现方法如下：

```
ASCII   DB   30H,31H,,32H,33H,34H, 35H,36H,37H,38H,39H ;ASCII 表格
             MOV AL,5                                  ;取“5”的偏移量
             MOV BX,OFFSET ASCII       ;取表首地址
             XLAT                                      ;查表转换
```

结果“5”的 ASCII 码在 AL 中，即(AL)＝35H。

BX 寄存器中包含着表的首地址，所在的段由隐含值确定。但也允许段超越，此时必须在指令中写明重设的段寄存器。XLAT 指令的几种表示形式如下：

XLAT ； 不写操作数

XLAT src-table ； 写操作数

XLATB ； B 表示字节类型，不允许再写操作数

XLAT ES：src-table ； 重设段寄存器为 ES

2. 输入输出指令

输入输出指令共有两条。输入指令 IN 用于从外设端口接收数据，输出指令 OUT 则向端口发送数据。无论是接收到的数据或是准备发送的数据都必须在累加器 AL(字节)或 AX(字)中，所以这是两条累加器专用指令。

输入输出指令的寻址可分为两大类：一类是端口直接寻址的输入输出指令；另一类是端口通过 DX 寄存器间接寻址的输入输出指令。在直接寻址的指令中只能寻址 256 个端口(0～255)，而间接寻址的指令中可寻址 64k 个端口(0～65535)。

1) 输入指令

(1) 直接寻址的输入指令。

指令格式及操作：

IN acc，port ；acc←(port)

此指令是将 8/16 位数据直接经输入端口 port(地址 0～255)送入 AL/AX 累加器中。

(2) 间接寻址的输入指令。

指令格式及操作：

IN acc，DX ；acc←(DX)

此指令是从 DX 寄存器内容指定的端口中，将 8/16 位数据送入 AL/AX 累加器中。这种寻址方式的端口地址由 16 位地址表示，执行此指令前应将 16 位地址存入 DX 寄存器中。

2) 输出指令

(1) 直接寻址的输出指令。

指令格式及操作：

OUT port， acc ；port←(acc)

此指令是从 AL(8 位)或 AX(16 位)累加器输出 8/16 位数据到指令指定的 I/O 端口中。

(2) 间接寻址的输出指令。

指令格式及操作：

OUT DX， acc ；(DX)←(acc)

此指令是从 AL(8 位)或 AX(16 位)累加器中输出 8/16 位数据到由 DX 寄存器内容指定的 I/O 端口中。

3. 目的地址传送指令

8086 CPU 提供了三条把地址指针写入寄存器或寄存器对的指令，它们可以用来写入近地址指针和远地址指针。

1) 取有效地址指令

指令格式：

LEA reg16，mem

LEA 是将一个近地址指针写入到指定的寄存器。指令中的目标操作数必须是一个 16 位的通用寄存器，源操作数必须是一个存储器操作数，指令的执行结果是把源操作数的有效地址，即 16 位的偏移地址源传送到目标寄存器。例如：

LEA BX，BUFFER

LEA AX，[BP][DI]

LEA DX，BETA[BX][SI]

注意 LEA 指令与 MOV 指令的区别，比较下面两条指令：

LEA BX，BUFFER

MOV BX，BUFFER

前者将存储器 BUFFER 的有效地址送到 BX，而后者是将 BUFFER 的内容送到 BX。以下两条指令功能相同：

LEA BX，BUFFER

MOV BX， OFFSET BUFFER

其中 OFFSET BUFFER 表示存储器 BUFFER 偏移地址。

2) 地址指针装入 DS 指令

指令格式：

LDS regl6，mere32

LDS 指令和下面即将介绍的 LES 指令都是用于写入远地址指针，源操作数可以是任意存储器，目标操作数是任意 16 位通用寄存器。LDS 传送 32 位远地址指针，前者送指定寄存器，后者送数据段寄存器 DS。例如：

LDS SI，[0010H]

设原来 DS＝C000H，而有关存储单元的内容为(C0010H)＝80H，(C0011H)＝01H，(C0012H)＝00H，(C00l3H)＝20H，则执行以上指令后，SI 寄存器的内容为 0180H，段寄存器 DS 的内容为 2000H。

3) 地址指针装入 ES 指令

指令格式：

LES regl6，mere32

LES 指令与 LDS 类似，也是装入一个 32 位的远地址指针，偏移量送到指定寄存器，段基值送到附加段寄存器 ES。

目的地址传送指令常常用于在串操作时建立初始的地址指针。

4. 标志传送指令

CPU 中有一标志寄存器 FLAG，其中每一状态标志位表示 CPU 运行的状态。许多指令执行结果会影响标志寄存器的某些状态标志位。同时，有些指令的执行也受标志寄存器中某些位的控制。标志传送指令共四条，均是单字节指令，指令的操作数以隐含的形式存在(隐含的操作数是 AH 寄存器)。

1) 取标志指令

指令格式：

LAHF

LAHF 指令将 FLAG 中的五个标志位传送到累加器 AH 的对应位，如图 5.29 所示。LAHF 指令对状态标志位没有影响

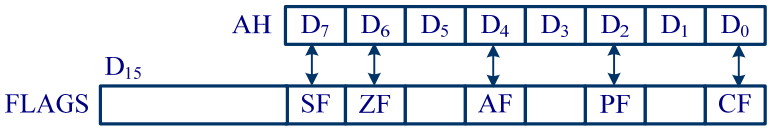


图 5. 29　LAHF 和 SAHF 指令操作示意图

2) 置标志指令

指令格式：

SAHF

SAHF 指令的传送方向与 LAHF 相反，将 AH 寄存器中的第 7、6、4、2、0 位分别传送到标志寄存器对应位的状态，但其余状态标志位即 OF、DF、IF 和 TF 不受影响，如图 5.29 所示。

3) 标志压入堆栈指令

指令格式及操作：

PUSHF；SP←(SP)-2，(SP)＋l：(SP)←(FLAG)

PUSHF 指令先将 SP 减 2，然后将标志寄存器 FLAG 的内容(16 位)压入堆栈。这条指令对状态标志位没有影响。

4) 标志弹出堆栈指令

指令格式及操作：

POPF；　FLAG←((SP)＋1，(SP))，SP←(SP)＋2

POPF 指令的操作与 PUSHF 相反，它将堆栈内容弹出到标志寄存器，然后 SP 加 2。POPF 指令对状态标志位有影响。

PUSHF 和 POPF 指令可用于保护调用过程以前标志寄存器的值，过程返回以后再恢复这些标志状态，或用来修改标志寄存器中相应标志位的值。

二、算术运算指令

8086 的算术运算指令可处理四种类型的数：无符号的二进制数、带符号的二进制数、无符号压缩十进制数(压缩 BCD 码)、无符号非压缩十进制数(非压缩 BCD 码)。压缩十进制数只有加/减运算，其他类型的数均可进行加、减、乘、除运算。

8086 提供了各种调整操作指令，因此除了可以对二进制数进行算术运算以外，也可以方便地进行压缩的或非压缩的十进制数的算术运算。

8086 的算术运算指令将运算结果的某些特性传送到 6 个标志上去，这些标志中的绝大多数可由跟在算术运算指令后的条件转移指令进行测试，以改变程序的流程。因而掌握指令执行结果对标志的影响，对编程有着重要的作用。

除了符号扩展指令，其余均影响标志位。

1. 加法指令

加法指令包括不带进位加法指令、带进位加法指令和加 1 指令。

1) 加法指令

指令格式及操作：

ADD dst，src；(dst)←(dst)＋(src)

ADD 将目标操作数与源操作数相加，结果存入目标操作数，影响标志寄存器。ADD 指令的操作数类型与 MOV 指令类似，但段寄存器不参与运算。例如：

ADD CL，1

ADD DX，SI

ADD AX，MEM

ADD DATA[BX]，AL

ADD ALP [DI]，30H

相加的数据类型可以根据编程者的意图，规定为带符号数或无符号数，如果相加结果超出范围，则发生溢出。例如：

MOV AL，7EH

MOV BL，5BH

ADD AL，BL

执行以上三条指令以后，相加结果(AL)＝D9H，此时各状态标志位的状态为：SF＝1，ZF＝0，AF＝1，PF＝0，CF＝0，OF＝l。其中 OF＝1 表示发生了溢出，这是由于相加结果超过了 127。但最高位并未产生进位，故 CF＝0。

2) 带进位加法指令

指令格式及操作：

ADC dst，src ； (dst)←(dst)＋(src)＋(CF)

ADC 指令是将目标操作数与源操作数相加，再加上进位标志 CF 的内容，然后将结果送目标操作数。操作数的类型与 ADD 指令相同，而且 ADC 指令同样也可以进行字节操作或字操作。

带进位加法指令主要用于多字节数据的加法运算。如果低字节相加时产生进位，则在下一次高字节相加时将这个进位加进去。

要求计算两个多字节十六进制数之和：3B74AC60F8H＋20D59E36C1H＝？式中被加数和加数均有五个字节，可以编一个循环程序实现以上运算。假设已将被加数和加数分别存入从 DATAl 和 DATA2 开始的两个内存区，且均为低位字节在前，高位字节在后，如图 3.8 所示。要求相加所得结果仍存回以 DATAl 为首址的内存区。

源程序如下：

MOV CX，5 ； 设置循环次数

MOV SI，0 ； 置位移量初值

CLC ； 清进位

LOOPER： MOV AL，DATA2[SI] ； 取另一个加数

ADC DATAl[SI]，AL ； 和另一个加数相加

INC SI ； 位移量加 1

DEC CX ； 循环次数减 1

JNZ LOOPER ； 加完否，若没完，转 LOOPER，继续相加

HLT ； 程序暂停

3) 加 1 指令

指令格式及操作：

INC dst ； dst←(dst)＋1

INC 指令将目标操作数加 1。指令影响 SF、ZF、AF、PF、OF，但对 CF 没影响。INC 指令的目标操作数可以是寄存器或存储器，但不能是段寄存器。其类型为字节操作或字操作均可。例如：

INC DL

INC SI

INC BYTE PTR [BX][SI]

INC WORD PTR [DI]

指令中的 BYTE PTR 或 WORD PTR 分别指定随后的存储器操作数的类型是字节或字。INC 指令常常用于在循环程序中修改地址。

2. 减法指令

减法指令包括不带借位减法指令、带借位减法指令、减 1 指令、求补指令和比较指令。

1) 减法指令

指令格式及操作：

SUB dst，src；dst←(dst)-(src)

SUB 指令用目标操作数减源操作数，结果送回目标操作数。该指令对状态标志位有影响。操作数的类型与加法指令一样，即目标操作数可以是寄存器或存储器，源操作数可以是立即数、寄存器或存储器，但不允许两个存储器相减。既可以字节相减，也可以字相减。例如：

SUB AL，37H，

SUB DX，BX

SUB CX，VAREl

SUB ARRAY[DL]，AX

SUB BETA[BX][DL]，512H

减法数据的类型也可以根据程序员的要求，约定为带符号数或无符号数。

2) 带借位的减法指令

指令格式及操作：

SBB dst，src；ds←(dst)-(src)-(CF)

SBB 指令是将目标操作数减源操作数，然后再减进位标志 CF，并将结果送回目标操作数，SBB 指令对标志的影响与 SUB 指令相同。

目标操作数及源操作数的类型也与 SUB 指令相同。8 位或 16 位数均可运算。例如：

SBB BX，l000H

SBB CX，DX

SBB AL，DATAl

SBB DISP[BP]，BL

SBB [SI＋6]，97H

带借位减指令主要用于多字节的减法。

3) 减 1 指令

指令格式及操作：

DEC dst，dst←(dst)–1

DEC 指令将目标操作数减 1。指令对状态标志位 SF、ZF、AF、PF 和 OF 有影响，但不影响进位标志 CF。操作数与 INC 一样，可以是寄存器或存储器(段寄存器不可)。其类型是字节操作或字操作均可。例如：

DEC BL

DEC AX

DEC [BX]

DEC WORD PTR [BP][DL]

在循环程序中常常利用 DEC 指令来修改循环次数。例如：

MOV AX，0FFFFH

CYC： DEC AX

JNZ CYC

HLT

以上程序段中，DEC AX 指令重复执行 65535 次，此程序是一段延时程序。

4) 求补指令

指令格式及操作：

NEG dst ； dst←0-(dst)

NEG 指令的操作是用“0”减去目标操作数，结果送回原来的目标操作数。对状态标志位有影响。可以对 8 位数或 16 位数求补，实际为求负。例如：

NEG BL

NEG AX

NEG BYTE PTR [BP][SI]

NEG WORD PTR [DI＋20]

利用 NEG 指令可以得到负数的绝对值。

内存数据段存放了 100 个带符号字节型数据，首地址为 AREA1，要求将各数取绝对值后存入以 AREA2 为首址的内存区。

由于 100 个带符号数中可能既有正数，又有负数，因此先要判断正负。如为正数，可以原封不动地传送到另一内存区；如为负数，则需先求补即可得到负数的绝对值，然后再传送。程序如下：

LEA SI，AREAl ； SI 为源地址指针

LEA DI，AREA2 ； DI 为目的地址指针

MOV CX，100 ； CX 为循环次数

CHECK ： MOV AL，[SI] ；取一个带符号数到 AL

OR AL，AL ； AL 内容不变，但使之影响标志

JNS NEXT ；若(SF)＝0，则转 NEXT

NEG AL ；否则求补

MOV [DI]，AL ；传送到目的地址

INC SI ；源地址加 1

INC DI ；目的地址加 1

DEC CX ；循环次数减 1

JNZ CHECK ；如不等于零，则转 CHECK

HLT ；停止

5) 比较指令

指令格式及操作：

CMP dst，stc ； dst-(src)

CMP 用目标操作数减源操作数，但结果不送回目标操作数。因此，执行比较指令 CMP 后，被比较的两个操作数内容均保持不变，而比较结果反映在标志寄存器中。这是 CMP 比较指令与 SUB 区别所在；比较指令目标操作数可以是寄存器或存储器，源操作数可以是立即数、寄存器或存储器，但不能同时为存储器。可以进行字节比较，也可以进行字比较。例如：

CMP AL，0AH ； 寄存器与立即数比较

CMP CX，DI ； 寄存器与寄存器比较

CMP AX，AREAl ； 寄存器与存储器比较

CMP [BX＋5]，SI ； 存储器与寄存器比较

CMP GAMM， l00 ； 存储器与立即数比较

3. 乘法指令

8086 指令系统中有两条乘法指令，可以实现无符号数的乘法和带符号数的乘法，它们只有源操作数，隐含目标操作数。CPU 在执行乘法时，一个操作数始终放在累加器中(8 位 AL；16 位 AX)，这是隐含的。8 位数相乘结果 16 位，存放在 AX 中，16 位数相乘结果 32 位，高 16 位存放在 DX 中，低 16 位存放在 AX 中。

1) 无符号数乘法指令

指令格式及操作：

MUL src ； AX←(src)×(AL) (字节乘法) ； DX：AX←(src)×(AX)(字乘法)

MUL 指令对状态标志位 CF、OF 有影响，SF、ZF、AF、PF 不确定。如果运算结果高位(AH 或 DX)为零，则状态标志位 CF＝OF＝0，否则 CF＝OF＝1，此时表示 AH 或 DX 中包含乘积的有效位。例如：

MUL AL ； AL 乘 AL

MUL BX ； AX 乘 BX

MUL BYTE PTR [DI＋6] ； AL 乘存储器(8 位)

MUL WORD PTR ALPHA ； AX 乘存储器(16 位)

MOV AL，14H ； (AL)＝14H

MOV CL，05H ； (CL)＝05H

MUL CL ； (AX)＝0064H，(CF)＝(OF)＝0

2) 带符号数的乘法

指令格式：

IMUL src ； AX←(src)×(AL) (字节乘法)；DX：AX←(src)×(AX)(字乘法)

IMUL 指令对状态标志位的影响以及操作过程均与 MUL 指令相同。但 IMUL 指令进行带符号数乘法，指令将两个操作数均认作带符号数，8 位和 16 位带符号数的取值范围分别是－128～＋127(字节)和－32768～＋32767。例如：

MOV AX，04E8H ；(AX)＝04E8H

MOV BX，4E20H ；(BX)＝4E20H

IMULBX ；(DX：AX)＝(AX)×(BX)

以上指令的执行结果(DX)＝017FH，(AX)＝4D00H，且(CF)＝(OF)＝1。实际上，以上指令完成带符号数(＋1256)和(＋20000)的乘法运算，得到乘积为(＋25120000)。由于此时 DX 中结果的高半部分包含着乘积的有效数字，故标志位(CF)＝(OF)＝1。

4. 除法指令

8086CPU 执行除法时规定：除数只能是被除数的一半字长。当被除数为 16 位时，除数应为 8 位，被除数为 32 位时，除数应为 16 位。并规定：

(1) 当被除数为 16 位，应存放于 AX 中。除数为 8 位，可存放在寄存器/存储器中。而得到的 8 位商放在 AL 中，8 位余数放在 AH 中。

(2) 当被除数为 32 位，应存放于 DX 和 AX 中。除数为 l6 位，可存放在寄存器/存储器中。而得到的 16 位商放在 AX 中，16 位余数放在 DX 中。

8086 指令系统中有两条除法指令，它们是无符号数除法指令和带符号数的除法指令。

1) 无符号数除法指令

指令格式及操作：

DIV src ；AL←(AX)/(src)的商(字节除法)，AH←(AX)/(src)的余数

；或 AX←(DX：AX)/(src)的商(字除法)，DX←(DX：AX)/(src)的余数

在字节除法中，AX 除以 src，被除数为 16 位，除数为 8 位。执行 DIV 指令后，商在 AL，余数在 AH 中；字除法中，DX、AX 除以 src，被除数为 32 位，除数为 16 位，除的结果，商在 AX，余数在 DX 中。执行 DIV 指令时，如果除数为 0，或字节除法时，AL 寄存器中的商大于 FFH，或字除法时，AX 寄存器中的商大于 FFFFH，CPU 立即自动产生类型号为 0 的中断。DIV 指令对状态标志位 CF、OF、SF、ZF、AF、PF 的影响不确定。

DIV BL ；AX 除以 BL

DIV CX ；(DX：AX) 除以 CX

DIV WORD PTR ALPHA ；(DX：AX) 除以存储器

如果被除数和除数的字长相等，可以在用 DIV 指令进行无符号数除法之前，将被除数的高位扩展 8 个零或 16 个零。

2) 带符号除法指令

指令格式及操作：

IDIV src ；AL←(AX)/(src)的商(字节除法)，AH←(AX)/(src)的余数

；或 AX←(DX：AX)/(src)的商(字除法)，DX←(DX：AX)/(src)的余数

执行 IDIV 指令时，如果除数为0，或字节除法时，AL 寄存器中的商超出(-128～＋127)，或字除法时，AX 寄存器中的商超出(-32768～＋32767)，CPU 立即自动产生类型号为 0 的中断。IDIV 指令对状态标志位 CF、OF、SF、ZF、AF、PF 的影响不确定。

5. 符号扩展指令

符号扩展指令包括字节扩展指令和字扩展指令。

1) 字节扩展指令

指令格式及操作：

CBW ；如果(AL)＜80H，则(AH)=00H，否则(AH)=FFH

CBW 指令将一个字节(8 位)，按其符号扩展成字，它是隐含操作数指令，隐含操作数为 AL 和 AH，对状态标志位没有影响。例如：

(1) MOV AL，4FH

CBW ；(AH)=00000000B

(2) MOV AL，0F4H

CBW ；(AH)=11111111B

2) 字扩展指令

指令格式及操作：

CWD

CWD 指令将一个字(16 位) 按其符号扩展成双字(32 位)，它是隐含操作数指令，隐含的操作数为寄存器 AX 和 DX 中的值。CWD 指令与 CBW 一样，对状态标志位没有影响。

CBW 和 CWD 指令在带符号数的乘法(IMUL)和除法(IDIV)运算中十分有用，常常在字节或字的运算之前，将 AL 和 AX 中数据的符号位进行扩展。例如：

MOV AL，MUL-BYTE ；(AL)←8 位被乘数(带符号数)

CBW ；扩展成为 16 位带符号数，在 AX 中

IMUL RSRC-WORD ；两个 16 位带符号数相乘，结果在 DX：AX 中

6. 十进制数(BCD 码)运算指令

以上介绍的是二进制数的算术运算。二进制数在计算机上进行运算是非常简单的。但是，通常人们习惯于用十进制数。在计算机中，十进制数是用 BCD 码来表示的。BCD 码有两类：压缩十进制数(压缩 BCD 码)和无符号非压缩十进制数(非压缩 BCD 码)，8086 用 BCD 码的运算指令算出结果，然后再用专门的指令对结果进行修正(调整)，使之转变为 BCD 码表示的正确的结果。

下面举例说明 BCD 码运算为什么要调整以及怎样进行调整：

12+23=35

```
        0001  0010
  +     0010  0011
  ---------------
        0011  0101
```

结果正确，这时调整指令不需要做什么。

19+23=35

```
        0001  1001
  +     0010  0011
  ---------------
        0011  1100
```

结果不正确，因为在进行二进制加法运算时，低 4 位向高 4 位有一个进位，这个进位是按十六进制进行的，即低 4 位逢十六才进一，而十进制数应是逢十进一。因此，比正确结果少 6，这时，调整指令应在低 4 位上加 6。即：

19+23=42

```
        0001  1001
  +     0010  0011
  ---------------
        0011  1100
  +     0000  0110
  ---------------
        0100  0010
```

由上面例子可知，加法运算后，低 4 位＞9，调整指令加 06H。同样，如加法运算后高 4 位＞9，调整指令加 60H。

1) 十进制加法的调整指令

根据 BCD 码的种类，对 BCD 码加法进行十进制调整的指令有两条：AAA 和 DAA。

(1) 非压缩型 BCD 码调整指令。

指令格式：

AAA

AAA 也称为加法的 ASCII 调整指令。指令后面不写操作数，但实际上隐含累加器操作数 AL 和 AH。指令的操作为：

如果 (AL)∧0FH＞9，或(AF)=1

则 AL←(AL)＋06H

AH←(AH)＋1

AF←1

否则 CF←(AF)

AL←((AL)∧0FH)

AAA 指令对 AF 和 CF，OF、SF、ZF、PF 的影响不确定。AAA 指令能对加法的结果 AL 的内容进行调整。

编程计算 4609+3875。

本例要求实现十进制多位数的加法，假设被加数的每一位数都以 ASCII 码形式；存放在内存中，低位在前，高位在后。另外留出 4 个存储单元，以便存放相加所得的结果，如图 3.12 所示。程序如下：

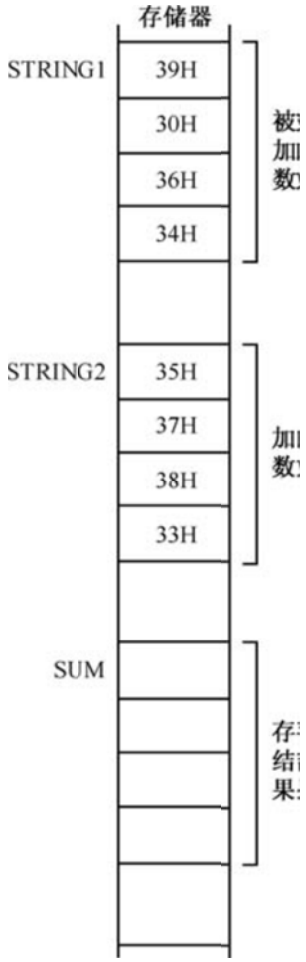


图 5. 30　数据存放情况

LEA SI ， STR1 ； (SI)←被加数地址指针

LEA BX ， STR2 ； (BX)←加数地址指针

LEA DI ， SUM ； (DI)←结果地址指针

MOV CX ， 4 ； (CX)←循环次数

CLC ； (CF)=0

N：MOV AL，[SI] ； 取一个字节被加数

ADC AL ， [BX] ； 与加数相加

AAA ； 调整

LEA SI ， STR1 ； (SI)←被加数地址指针

LEA BX ， STR2 ； (BX)←加数地址指针

LEA DI ， SUM ； (DI)←结果地址指针

MOV CX ， 4 ；(CX)←循环次数

CLC ；(CF)=0

N：MOV AL，[SI] ；取一个字节被加数

ADC AL ，[BX] ；与加数相加

AAA ；调整

(2) 压缩型 BCD 码调整指令。

指令格式：

DAA

DAA 指令同样不带操作数，实际上隐含寄存器操作数 AL。指令的操作为：

如果 (AL)∧0FH＞9，或(AF)=1

则 AL←(AL)+06H

AF←1

如果 (AL)∧0F0H＞9FH，或(CF)＝1

则 AL←(AL)＋60H

CF←1

DAA 指令影响 AF、CF、SF、ZF、PF，但不影响 OF。DAA 指令只对加法的结果 AL 的内容进行调整，不影响 AH。例：

MOV AL，68H ；(AL)＝68H

MOV BL，59H ；(BL)＝59H

ADD AL，BL ；(AL)＝C1H，(AF)＝1

DAA ；(AL)＝27 H ，(CF)＝1

2) 十进制减法的调整指令

同加法一样，对 BCD 码减法进行十进制调整的指令也有两条：AAS 和 DAS。

指令格式：

AAS

AAS 也称为减法的 ASCII 码的调整指令。隐含寄存器操作数为 AL 和 AH。对非压缩型 BCD 码调整。指令的操作为：

如果 (AL)∧0FH＞9，或(AF)＝1

则 AL←(AL)－06H

AH←(AH)－1

AF←1

CF←(AF)

否则 AL←((AL)∧0FH)

AAS 指令影响 AF 和 CF，OF、SF、ZF、PF 不确定。

编程计算 15　4。先将被减数和减数以压缩的 BCD 码的形式分别存放在 AH(被减数的十位)、AL(被减数的个位)和 BL(减数)中，然后用 SUB 指令进行减法，再用 AAS 指令进行调整。可用以下指令实现：

MOV AX，0105H ；(AH)＝01H，(AL)＝05H

MOV BL，04H ；(BL)＝04H

SUB AL，BL ；(AL)＝03H－04H＝FFH

AAS ；(AL)＝09H，(AH)＝0

以上指令的执行结果为 15－4＝11，此结果仍以非压缩型 BCD 码的形式存放，个位在 AL 寄存器，十位在 AH 寄存器。

3) 压缩型 BCD 码调整指令

指令格式：

DAS

指令对减法进行十进制调整，指令隐含寄存器操作数 AL。在减法运算时，DAS 指令对压缩型 BCD 码进行调整，其操作为

如果 ((AL)∧0FH)＞9 或(AF)＝1

则　　AL←(AL)－06H

AF←1

如果 (AL)＞9FH 或(CF)＝1

则　　AL←(AL)－60H

CF←1

与 DAA 类似，DAS 指令影响 AF、CF、SF、ZF、PF，但不影响 OF。DAS 指令只对减法的结果 AL 的内容进行调整，任何时候都不影响 AH。

编程计算 83－38。采用压缩型 BCD 码存放原始数据，则以上减法运算可用下列几条指令实现：

MOV AL，83H ；(AL)＝83H

MOV BL，38H ；(BL)＝38H

SUB AL，BL ；(AL)＝4BH

DAS ；(AL)＝45H

4) 十进制乘除法的调整指令

对于十进制数的乘除法运算，8086/8088 指令系统只提供了非压缩型 BCD 码的调整指令，而没有提供压缩型 BCD 码的调整指令。因此，8086/8088 CPU 不能直接进行压缩型 BCD 码的乘除法运算。

非压缩型 BCD 码的乘除法与加减法相同，加减法可以直接用 ASCII 码参加运算，而不管其高位上有无数字，只要在加减指令后用一条非压缩型 BCD 码的调整指令，就能得到正确结果。而乘除法要求参加运算的两个数高 4 位是 0 的非压缩型 BCD 码，低 4 位是十进制数。也就是说，如果用 ASCII 码进行非压缩型 BCD 码的乘法运算，在乘除法运算之前，必须将高 4 位清零。

(1) 非压缩型 BCD 码的乘法调整指令。

指令格式：

AAM

AAM 指令也是一个隐含了寄存器操作数 AL 和 AH 的指令。在乘法运算时，调整之前，先用 MUL 指令将两个真正的非压缩型的 BCD 码相乘，结果放在 AX 中。然后用 AAM 指令对 AL 寄存器进行调整，于是在 AX 中就可得到正确的非压缩型 BCD 码的结果，其乘积的高位在 AH 中，乘积的低位在 AL 中。AAM 指令的操作为：

AH←(AL)/0AH 的商；即 AL 除以 10，商送 AH

AL←(AL)/0AH 的余数；即 AL 除以 10，余数送 AL

AAM 指令的操作实质上是将 AL 寄存器中的二进制数转换成为非压缩型的 BCD 码，十位存放在 AH 寄存器，个位存放在 AL 寄存器。AAM 指令执行以后，将根据 AL 中的结果影响状态标志位 SF、ZF 和 PF，但 AF、CF 和 OF 的值不定。

要求进行以下十进制乘法运算：7×9＝？可编程序段如下：

MOV AL，07H ；(AL)＝07H

MOV BL，09H ；(BL)＝09H

MUL BL ；(AX)＝07H×09H＝003FH

AAM ；(AH)＝06H，(AL)＝03H，(SF)＝0，(ZF)＝0，(PF)＝l

已知 7×9＝63。以上指令执行以后，十进制乘积也是以非压缩型 BCD 码的形式存放在 AX 中。由于(AL)＝03H(0000000llB)，故决定了(SF)=0，(ZF)=0，(PF)=l。

(2) 非压缩型 BCD 码的除法调整指令。

指令格式：

AAD

AAD 指令也是一个隐含了寄存器操作数 AL 和 AH 的指令，它是对非压缩型 BCD 码进行调整，其操作为：

AL←(AH)×0AH＋(AL)

AH←0

即将 AH 寄存器的内容乘以 10 并加上 AL 寄存器的内容，结果送回 AL，同时将零送 AH。以上操作实质上是将 AX 寄存器中非压缩型 BCD 码转换成为真正的二进制数，并存放在 AL 寄存器中。执行 AAD 指令以后，将根据 AL 中的结果影响状态标志位 SF、ZF 和 PF，但其余几个状态标志位，如 AF、CF 和 OF 的值则不确定。

AAD 指令的用法与其他非压缩型 BCD 码调整指令(如 AAA、AAS、AAM)有所不同。AAD 指令使用在除法指令之前进行调整，方可得到正确的非压缩型 BCD 码的结果。

编程进行以下十进制除法运算，73÷2=？可先将被除数和除数以非压缩型 BCD 码的形式分别存放在 AX 和 BL 寄存器中，被除数的十位在 AH，个位在 AL，除数在 BL。

先用 AAD 指令对 AX 中的被除数进行调整，然后进行除法运算，并对商进行再调整。可编成如下程序段：

MOV AX，0703H ；(AH)＝07H，(AL)＝03H

MOV BL，02H ；(BL)＝02H

AAD ；(AL)＝49H(十进制数 73)

DIV BL ；(AL)＝24H(商)，(AH)＝01H(余数)

AAM ；(AH)＝03H，(AL)＝06H

已知 73÷2=36…1。以上几条指令执行的结果为，在 AX 中得到非压缩型 BCD 码的商，但余数已被丢失。如果需要保留余数，则应在 DIV 指令之后，用 AAM 指令调整之前，将余数暂存到寄存器中。如果有必要，还应设法对余数也进行调整。

三、逻辑运算指令

8086 逻辑运算指令有 AND 逻辑“与”、TEST 测试、OR 逻辑“或”、XOR 逻辑“异或”、NOT 逻辑“非”运算。以上指令只有 NOT 逻辑“非”指令对状态标志寄存器没有影响。其他指令根据各自的逻辑运算结果影响 SF、ZF、 PF，将 CF、OF 清 0，AF 的值不确定。

1. 逻辑“与” 指令

指令格式及操作：

AND dst，src ；dst←(dst)∧(src)

AND 将源操作数与目标操作数按位进行“与”运算，结果送回目标操作数。两个操作数不能同时为存储器。例如：

AND AL，00001111H ；寄存器“与”立即数

AND CX，DI ；寄存器“与”寄存器

AND SI，MEM ；寄存器“与”存储器

AND ALP [DI]，AX ；存储器“与”寄存器

AND [BX][SI]，0FFFEH ；存储器“与”立即数

2. TEST 测试

TEST 指令的操作实质上与AND 指令相同,即把目标操作数和源操作数进行逻辑“与”。二者的区别在于 TEST 指令不把逻辑运算的结果送回目标操作数，因此两个操作数的内容均保持不变，即目标操作数将不被破坏。逻辑“与”的结果反映在状态标志位上，例：“与”的最高位是 0 还是 1，结果是否全为 0，结果中 1 的个数是奇数还是偶数分别由 SF、ZF、PF 体现。将 CF、OF 清 0，AF 的值不确定。例如：

TEST BH，7 ；寄存器“与”立即数(结果不回送，下同)

TEST SI，BP ；寄存器“与”寄存器

TEST [SI]，CH ；存储器“与”寄存器

TEST [BX][DI]，6ACEH；存储器“与”立即数

TEST 指令常用于位测试，它与条件转移指令一起，共同完成对特定位状态的判断，并实现相应的程序转移。这种作用与比较指令 CMP 有些类似，不过 TEST 指令只比较某一指定的位，而 CMP 指令比较整个操作数(字节或字)。

3. 逻辑“或”指令

指令格式及操作：

OR dst，src；dst←dst∨(src)

OR 指令将目的操作数和源操作数按位进行逻辑“或”运算，并将结果送回目标操作数。OR 指令操作数的类型与 AND 指令相同，即目标操作数可以是寄存器或存储器，源操作数可以是立即数、寄存器或存储器，但不能同时都是存储器。例如：

OR BL，0F6H ；寄存器“或”立即数

OR AH，BL ；寄存器“或”寄存器

OR CL，BETA[BX][DI] ；寄存器“或”存储器

OR [BX][DI]， 80 H ；存储器“或”立即数

OR 指令的用途是将寄存器或存储器中的某些位置 1，而不管这些位原来的状态如何，并保持其他位状态不变。“或”指令将要求保持不变的位和“0”进行逻辑“或”，“或”指令将要求置 1 的位和“1”进行逻辑“或”，该指令影响 SF、ZF、PF。

AND 指令和 OR 指令有一个共同的特性：如果将一个寄存器的内容和该寄存器本身进行逻辑“与”操作或者逻辑“或”操作，则保持寄存器的内容不变，但影响 SF、ZF、PF。

4. XOR 逻辑“异或”指令

指令格式及操作：

XOR dst，src ； dst←dst⊕(src)

XOR 指令将目标操作数和源操作数按位进行逻辑“异或”运算，并将结果送回目标操作数。XOR 指令的操作类型与 AND 指令和 OR 指令相同，此处不再赘述，例如：

XOR DI，23F6H ；寄存器“异或”立即数

XOR SI，DX ； 寄存器“异或”寄存器

XOR CL，BUFFER ； 寄存器“异或”存储器

XOR MEM[BX]，AX ； 存储器“异或”寄存器

XOR TABLE[BP][SI]，3DH ； 存储器“异或”立即数

XOR 指令的用途是将寄存器或存储器中某些特定的位“求反”，而使其余位保持不变。为此，可将欲“求反”的位和“1”进行“异或”，而将要求保持不变的位和“0”进行“异或”。例如，若要使 AL 寄存器中第 1、3、5、7 位取反其他位不变，异或 10101010B(即 0AAH)即可。

MOV AL，0FH(AL)＝00001111B

XOR AL，0AAH；(AL)＝10100101B(0A5H)

XOR 指令的另一个用途将寄存器内容清零，例如：

XOR AX，AX ； AX 清零

XOR CX，CX ； CX 清零

而且，上述指令和 AND、OR 等指令一样，也将进位标志 CF 清零。XOR 指令的这种特性在多字节的累加程序中十分有用，它可以在循环程序开始前的初始化过程中使用。

5. 逻辑“非”运算

指令格式及操作

NOT dst ； dst←FFFFH-(dst) (字求反)

NOT 指令的操作数可以是 8 位或 16 位寄存器或存储器，但不能是立即数。

NOT AH ； 8 位寄存器求反

NOT CX ； 16 位寄存器求反

NOT BYTE PTR [BP] ； 8 位存储器求反

NOT WORD PTR COUNT ； 16 位存储器求反

四、移位指令

8086 指令系统的移位指令包括逻辑左移 SHL、算术左移 SAL、逻辑右移 SHR、算术右移 SAR 等非循环移位指令，还有循环移位指令，包括不带进位循环左移 ROL、循环右移 ROR 和带进位循环左移 RCL、循环右移 RCR。移位常数一定放在 CL 中。移位指令都影响状态标志位，但影响的方式各条指令不尽相同。

1. 非循环移位指令

1) 逻辑左移 SHL/算术左移 SAL

指令格式：

SHL dst，1

SAL dst，1

或　SHL dst，CL

SAL dst，CL

这两条指令的操作是将目标操作数顺序向左移 1 位或移 CL 位，左移 1 位时高位移入进位标志 CF，最低位补 0。指令操作示意图如图 5.31 所示，逻辑左移 SHL/算术左移 SAL 影响CF 和OF，如果移位次数是1，且移位后dst 最高位与CF 不相等，则溢出标志位OF=1，否则 OF=0。如果移位次数不是 1，则 OF 值不确定。OF 值表示移位是否改变符号位。例如：

SHL AH，1 ； 寄存器左移 1 位

SAL SI，CL ； 寄存器左移(CL)位

SAL WORD PTR [BX＋5]，1 ； 存储器左移 1 位

SHL DATA，CL ； 存储器左移寄存器(CL)中指定的位数

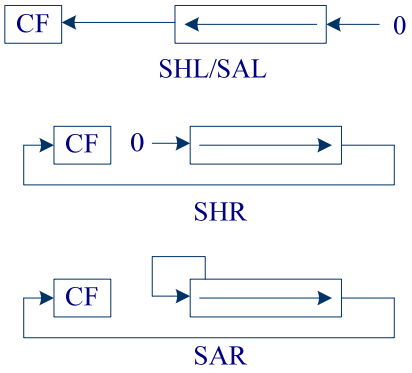


图 5. 31　非循环移位操作示意图

将一个 16 位无符号数乘以 l0。该数原来存放在以 FACTOR 为首地址的两个连续的存储单元中(低位在前，高位在后)。

因为 FACTOR×10＝(FACTOR×8)＋(FACTOR×2)，故可用左移指令实现以上乘法运算。编程如下：

MOV AX，FACTOR ； AX←被乘数

SHL AX，1 ； (AX)＝FACTOR×2

MOV BX，AX ； 暂存 BX

SHL AX，1 ； (AX)＝FACTOR×4

SHL AX，1 ； (AX)＝FACTOR×8

ADD AX，BX ； (AX)＝FACTOR×l0

HLT

以上程序的执行时间大约需 26 个周期。如用乘法指令编程，执行时间将超过 130 个时钟。

2) 逻辑右移指令

指令格式：

SHR dst，1/CL

SHR 指令的操作是将目标操作数顺序向右移 1 位或右移由 CL 寄存器指定的位数。逻辑右移 1 位时，低位移入进位标志 CF，最高位补 0。指令操作如图 5.31 所示。

SHR 指令也将影响 CF 和 OF 状态标志位。如果移位次数等于 1，且移位以后新的最高位与次高不相等，则溢出标志位 OF＝1，否则 OF＝0。OF 值表示移位是否改变符号位。例如：

SHR BL，1 ； 寄存器逻辑右移 1 位

SHR AX，CL ； 寄存器逻辑右移(CL)位

SHR BYTE PTR [DI＋BP]，1 ； 存储器逻辑右移 1 位

SHR WORD PTR BLOCK，CL ； 存储器逻辑右移(CL)位

将一个 16 位无符号数除以 512。该数原来存放在以 ABC 为首地址的两个连续的存储单元中。

因为 ABC÷512＝(DDA÷2)÷256，因此可用逻辑右移指令完成上述除法运算。编程如下：

MOV AX，ABC ； AX←被除数

SHR AX，1 ； (AX)＝DDA÷2

XCHG AL，AH ； (AL)←→(AH)，相当循环右移 8 位

CBW ； 清 AX 的高 8 位，(AX)＝DDA

HLT

当然，也可以将立即数 9 传送到 CL 寄存器，然后用指令(SHR AX，CL)完成除以 512 的运算。但是相比之下，上面的程序执行速度更快。

3) 算术右移指令

指令格式：

SAR dst，l/CL

SAR 指令的操作数与逻辑右移指令 SHR 有点类似，将目标操作数向右移 1 位或由 CL 寄存器指定的位数。逻辑右移 1 位时，低位移入进位标志 CF，最高位保持不变。算术右移指令对状态标志位 CF、OF、PF、SF 和 ZF 有影响，但 AF 的值不确定。例如：

SAR AX，1 ； 寄存器算术右移 1 位

SAR DX，CL ； 寄存器算术右移(CL)位

SAR [SI]，1 ； 存储器算术右移 1 位

SAR [BX]，CL ； 存储器算术右移(CL)位

2. 循环移位指令

8086 指令系统有四条循环移位指令，即不带进位标志 CF 的左循环移位指令 ROL 和右循环移位指令 ROR(也称小循环)，以及带进位标志 CF 的左循环移位指令 RCL 和右循环移位指令 RCR(也称大循环)。它们的操作如图 5.32 所示。

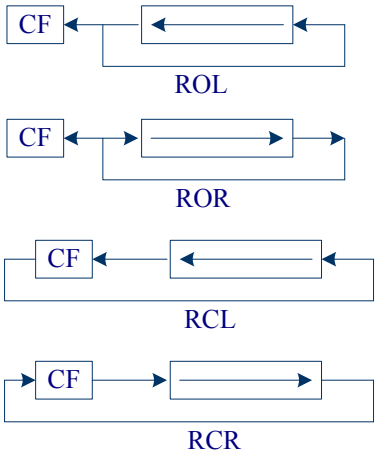


图 5. 32　循环移位指令操作示意图

循环移位指令的操作数类型与移位指令相同，可以是 8 位或 16 位的寄存器或存储器。指令中指定的左移或右移的位数也可以是 1 或由 CL 寄存器指定。但不能是 1 以外的常数或 CL 以外的寄存器。循环移位指令都只影响进位标志 CF 和溢出标志 OF，但 OF 标志的含义对于左循环移位指令和右循环移位指令有所不同。

1) 循环左移指令

指令格式：

ROL dst，1/CL

ROL 指令将目标操作数向左循环移动 1 位或 CL 寄存器指定的位数。最高位移到进位标志 CF，同时，最高位移到最低位形成循环，进位标志 CF 不在循环回路之内。

ROL 指令将影响 CF 和 OF 两个状态标志位。如果循环移位次数等于 1，且移位以后目的操作数新的最高位与 CF 不相等，则(OF)＝1，否则(OF)＝0。因此，OF 的值表示循环移位前后符号位是否有所变化。如果移位次数不等于 1，则 OF 的值不确定。例如：

ROL BH，1 ；寄存器循环左移 1 位

ROL DX，CL ；寄存器循环左移(CL)位

ROL [DI]，1 ；存储器循环左移 1 位

ROL HA，CL ；存储器循环左移(CL)位

2) 循环右移指令

指令格式：

ROR dst，1/CL

ROR 指令将目标操作数向右循环移动 1 位或右移由 CL 寄存器指定的位数。最低位移到进位标志 CF，同时最低位移到最高位 ROR 指令也将影响状态标志位 CF 和 OF。如果移位次数等于 1，且移位以后新的最高位与次高位不相等，则溢出标志位 OF＝1；否则 OF＝0。OF 值表示移位是否改变符号位。

3) 带进位循环左移指令

RCL dst，1/CL

RCL 指令将目标操作数连同进位标志 CF 一起向左循环移动 1 位或由 CL 寄存器指定的位数。最高位移入 CF，而 CF 移入最低位。RCL 指令对状态标志位的影响与 ROL 指令相同。

4) 带进位循环右移指令

指令格式：

RCR dst，1/CL

RCR 指令将目标操作数连同进位标志 CF 一起向右循环移动 1 位或由 CL 寄存器指定的位数。最低位移到进位标志 CF，同时进位标志 CF 移到最高位。RCR 指令对状态标志位的影响与 ROR 指令相同。

值得指出的是，循环移位指令与非循环移位指令有所不同。循环移位指令进行循环移位操作之后，操作数中原来各位的信息不会丢失，只是移到了操作数中的其他位或进位标志上，必要时还可以恢复。

五、控制转移指令

8086/8088 指令系统提供了大量指令，用于控制程序的流程。这类指令包括转移指令、循环控制指令、过程调用指令和中断指令四类。

1. 转移指令

转移是一种将程序控制从一处改换到另一处的最直接的方法。在 CPU 内部，转移是通过将目的地址传送给 IP 来实现的。

转移指令包括无条件转移指令和条件转移指令。

1) 无条件转移指令

无条件转移指令的操作是无条件地将程序转移到指令中指定的目标地址。目标地址可以用直接的方式给出，也可以用间接的方式给出。无条件转移指令对状态标志位没有影响。

(1) 段内直接转移。

指令格式及操作：

JMP near1abel ；IP←(IP)＋disp(16 位)

指令的操作数是一个近标号，该标号在本段(或本组)内。指令汇编以后，计算出 JMP 指的下一条指令的地址到目的地址之间的 16 位相对位移量 disp。指令的操作是将指令指针寄存器 IP 的内容加上相对位移量 disp，代码段寄存器 CS 的内容不变，从而使控制转移到目的地址。相对位移量可正可负，一般情况下，它的范围在-32768～+32767 之间，故需用

2 个字节表示，加上一个字节的操作码，这种段内直接转移指令共有 3 个字节。例如：

JMP NEXT

AND AL，7FH

NEXT：XOR AL，7FH

其中，NEXT 是本段内的一个标号，汇编语言计算出下一条指令(即 AND AL，7FH)的地址与标号 NEXT 代表的地址之间的相对位移量，执行 JMP NEXT 指令时，将上述位移量加到 IP 上，于是执行 JMP 指令之后，接着就执行 XOR AL，7FH 指令，实现了程序的转移。

(2) 段内直接短转移。

指令格式及操作：

JMP Shortlabel ；IP←(IP)＋disp(8 位)

段内直接短转移指令的操作数是一个短标号。此时，相对位移量 disp 的范围在-128～+127 之间，只需用 1 个字节表示。段内直接短转移指令共有 2 个字节。如果已知下一条指令到目的地址之间的相对位移量在-128～+127 的范围内，则可在标号前写上运算符 SHORT，实现段内直接短转移。但是，对于一个段内直接转移指令，如果相对位移量的范围在-128～+127 之间，而且目标地址的标号已经定义(即标号先定义后引用，这种情况称为向后引用的符号)，那么即使标号没写上运算符 SHORT，汇编程序也能自动生成一个 2 字节的短转移指令。这种情况属于隐含的短转移。如果向前引用标号(即标号先引用，后定义)，则标号前应写上运算符 SHORT，否则，即使位移量的范围不超过 128～+127，汇编后仍会生成一个 3 字节的近转移指令。例如：

向后引用的标号可以不写运算符 SHORT

label: … ；先定义标号 label

⋮ ；相对位移量不超过-128～＋127

JMP label ；后引用标号 label

向前引用的标号应当写明运算符 SHORT

JMP SHORT label ；先引用标号 label

⋮ ；相对位移量不超过-128～＋127

label: … ；此处定义标号 label

(3) 段内间接转移。

指令格式及操作：

JMP regl6/meml6；IP←(regl6)/IP←(meml6)

指令的操作是一个 16 位的寄存器(regl6)或存储器(meml6)地址。存储器可用各种寻址方式。指令的操作是用指定的寄存器或存储器中的内容作为目标的偏移地址取代原来 IP 的内容，以实现程序的转移，由于是段内转移，故 CS 寄存器的内容不变。例如：

JMP BX　　；本指令执行后，(IP)=(BX)

JMP WORD PTR[BX+DI]

对上面第二条指令，设指令执行前：(DS)=2000H，(BX)=1000H，(DI)=2006H，(23006H)=6E00H，则指令执行后，(IP)=6E00H，指令执行的过程如图 5.33 所示。由于是段内转移，其范围一定在当前代码段内，所以 CS 的内容不变。

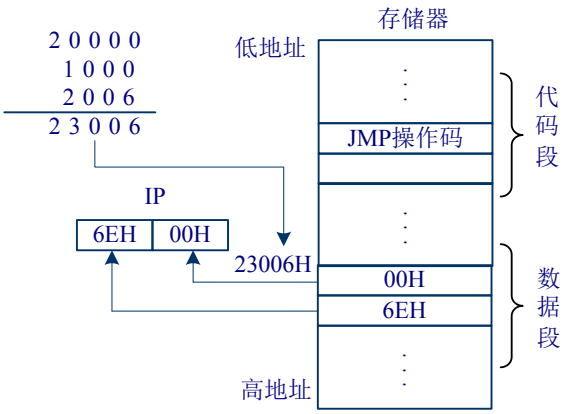


图 5.33　段内间接转移指令操作示意图

关于“PTR”的说明，请参阅本书第 6 章表达式部分关于分析运算符和合成运算符的介绍。

(4) 段间直接转移。

指令格式及操作：

JMP farlabel ；IP←OFFSET farlabel　　；CS←SEG farlabel

指令的操作数是一个远标号，该标号在另一个代码段内。指令的操作是将标号的偏移地址取代指令指针寄存器 IP 的内容，同时将标号的段基值取代段寄存器 CS 的内容，结果使控制转移到另一代码段内指定的标号处。例如：

JMP FAR PTR NEXT　　；远转移到 NEXT 处

JMP 8000H:0012H　　；(IP)←0012H，(CS)←8000H

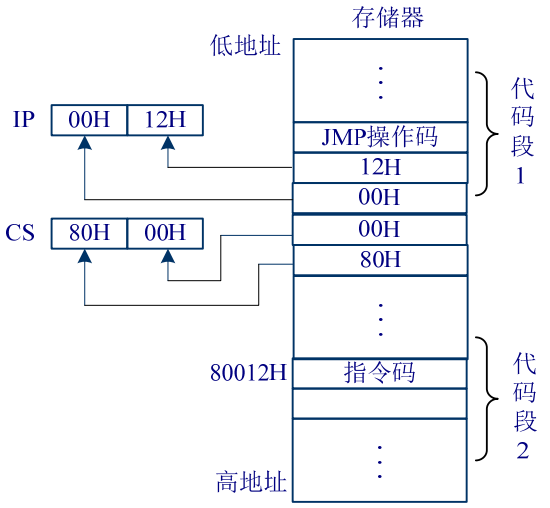


图 5. 34　段间直接转移操作示意

上述第二条指令的执行过程如图 5.34 所示。

(5) 段间间接转移。

指令格式及操作：

JMP mem32　；IP←(mere32)，CS←(mem32＋2)

指令的操作数是 32 位的存储器地址，指令的操作是将存储器的前两个字节送到 IP 寄存器，存储器的后两个字节送到 CS 寄存器，以实现到另一个代码段的转移。

JMP VARDOUBLEWORD

JMP DWORD PTR [BP][DI]

上面第一条指令中，VARDOUBLEWORD 应是一个已经定义成 32 位的存储器变量(例如可用数据定义伪指令 DD 定义)。第二条指令中，利用运算符 PTR 将存储器操作数的类型定义成 DWORD(双字，即 32 位)。例如：

JMP DWORD PTR[BX]

设指令执行前：(DS)=3000H，(BX)=3000H，(33000H)=12H，(33001H)=00H，(33002H)=00H，(33003H)=80H。则指令执行后：(IP)=0012H，(CS)=8000H。转移的目标地址=80012H。其操作示意如图 5.35 所示。

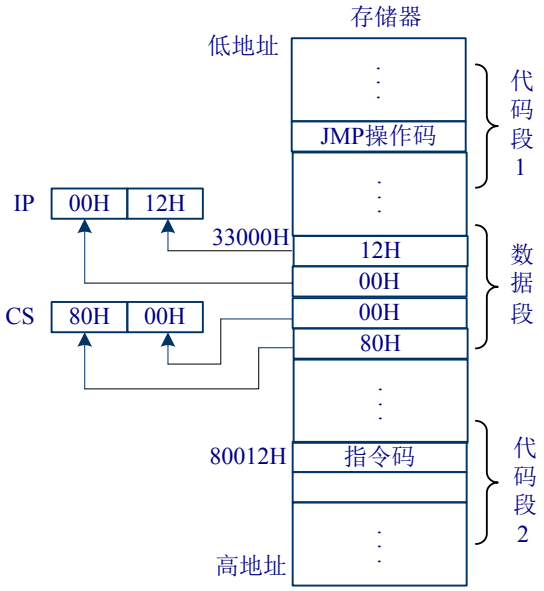


图 5. 35　段间间接转移操作示意

2) 条件转移指令

指令格式为：

JCC shortLabel

在汇编语言程序设计中，常利用条件转移指令来构成分支程序。指令助记符中的“CC”

表示条件。这种指令的执行包括两个过程：第一步，测试规定的条件；第二步，如果条件满足则转移到目的地址，否则，继续顺序执行。

条件转移指令也只有一个操作数，用以指明转移的目的地址。但是它与无条件转移指令 JMP 不同，条件转移指令的操作数必须是一个短标号，也就是说，所有的条件转移指令都是 2 字节指令，转移指令的下一条指令到目标地址之间的距离必须在-128～＋127 的范围内。如果指令规定的条件满足，则将这个位移量加到 IP 寄存器上，即 IP←(IP)＋disp，实现程序的转移。

绝大多数条件转移指令(除 JCXZ 指令外)将状态标志位的状态作为测试的条件。因此，首先应执行影响有关的状态标志位的指令，然后才能用条件转移指令测试这些标志，以确定程序是否转移。CMP 和 TEST 指令常与条件转移指令配合使用，因为这两条指令不改变目标操作数的内容，但可影响状态标志寄存器。

8086 的条件转移指令非常丰富，不仅可以测试一个状态标志位的状态，而且可以综合测试几个状态标志位；不仅可以测试无符号数的高低，而且可以测试带符号数的大小等，在编程时十分灵活、方便。下面将所有的条件转移指令的名称、助记符及转移条件列于表 5.17 中。同一行内用斜杠隔开的几个助记符，实质上代表同一条指令的几种不同的表示方法。

表 5.17 条件转移指令

指令名称	助记符	转移条件	备注
等于/零转移	JE/JZ	(ZF)＝1	
不等于/非零转移	JNE/JNZ	(ZF)＝1	
负转移	JS	(SF)＝1	
正转移	JNS	(SF)＝0	
偶转移	JP/JPE	(PF)＝1	
奇转移	JNP/JPO	(PF)＝0	
溢出转移	JO	(OF)＝1	
不溢出转移	JNO	(OF)＝0	
进位转移	JC	(CF)＝1	
不进位转移	JNC	(CF)＝0	
低于/不高于或不等于转移	JB/JNAE	(CF)＝1	无符号数
高于或等于/不低于转移	JAE/JNB	(CF)＝0	无符号数
高于/不低于或不等于转移	JA/JNBE	(CF)＝0 且(ZF)＝0	无符号数
低于或等于/不高于转移	JBE/JNA	(CF)＝1 或(ZF)＝1	无符号数
大于/不小于或不等于转移	JG/JNLE	(SF)＝(OF)且(CF)＝0	带符号数

大于或等于/不小于转移	JGE/JNL	(SF)＝(OF)	带符号数
小于/不大于或不等于转移	JL/JNGE	(SF)≠(OF)且(ZF)＝0	带符号数
小于或等于/不大于转移	JLE/JNG	(SF)≠(OF)或(ZF)＝1	带符号数
CX 等于零转移	JCXZ	(CX)＝0	

在内存的数据段中存放了若干个 8 位带符号数，数据块的长度为 COUNT(不超过 255)，首地址为 TABLE，试统计其中正元素、负元素及零元素的个数，并将个数分别存入 PLUS、MINUS 和 ZERO 单元。

为了统计正元素、负元素和零元素的个数，可先将 PLUS、MINUS 和 ZERO 三个单元清零，然后将数据表中的带符号数逐个取入 AL 寄存器并使其影响状态标志位，再利用前面介绍的 JS、JZ 等条件转移指令测试该数是一个负数、零还是正数，然后分别在相应的单元中进行计数。

编程如下：

```
XOR AL，AL ； AL←0
MOV PLUS，AL ； 清 PLUS 单元
MOV MINUS，AL ； 清 MINUS 单元
MOV ZERO，AL ； 清 ZERO 单元
LEA SI，TABLE ； SI←数据表首址
MOV CX，COUNT ； CX←数据表长度
CLD ； 清状态标志位 DF
CHECK：LODSB ； 取一个数据到 AL
OR AL，AL ； 使数据影响状态标志位
JS X1 ； 如为负，转 X1
JZ X2 ； 如为零，转 X2
INC PLUS ； 否则为正，PLUS 单元加 1
JMP NEXT
X1：INC MINUS ； MINUS 单元加 l
JMP NEXT
X2：INC ZERO ； ZERO 单元加 1
NEXT：LOOP CHECK ； CX 减 1，如不为零，则转 CHECK
HLT ； 停止
```

以上程序中的 LOOP CHECK 指令是一条循环控制指令，它的操作是将 CX 寄存器的内容减 1，如结果不等于零，则转移到短标号 CHECK。后面将讨论这条指令。

2. 循环控制指令

8086 指令系统专门设计了几条循环控制指令，用于使一些程序段反复执行，形成循环程序。循环控制指令有以下几条：

1) LOOP

指令格式：

LOOP shortlabel

LOOP 指令规定将 CX 寄存器作为计数器，指令的操作是先将 CX 的内容减 1，如结果不等于零，则转到指令中指定的短标号处；否则，顺序执行下一条指令。因此，在循环程序开始前，应将循环次数送 CX 寄存器。指令的操作只能是一个短标号，即跳转距离不超过-128～+127 的范围。LOOP 指令对状态标志位没有影响。LOOP 指令的应用可参阅前面的例子。

2) LOOPE/LOOPZ

指令格式：

LOOPE/ LOOPZ shortlabel

以上两种格式实际上代表同一条指令。本指令的操作是先将 CX 的内容减 1，如结果不等于零，且零标志(ZF)=1 则转移到指定的短标号。LOOPE/LOOPZ 指令对状态标志位也没有影响。

该指令是有条件地形成循环，即当规定的循环次数尚未完成时，还必须满足“相等”或者“等于零”的条件，才能继续循环。

3) LOOPNE/LOOPNZ

指令格式：

LOOPNE/LOOPNZ shortlabel

以上两种格式实际上代表同一条指令。本指令的操作是先将 CX 的内容减 1，如结果不等于零，且零标志(ZF)＝0 则转移到指定的短标号。LOOPNE/LOOPNZ 指令对状态标志位也没有影响。

3. 过程调用指令

如果有一些程序段需要在不同的地方多次反复地出现，则可以将这些程序段设计成为过程(相当于子程序)，每次需要时进行调用。过程结束后，再返回原来调用的地方。采用这种方法不仅可以使源程序的总长度大大缩短，而且有利于实现模块化的程序设计，使程序的编制、阅读和修改都比较方便。

被调用的过程可以在本段内(近过程)；也可在其他段(远过程)。调用的过程地址可以用直接的方式给出，也可用间接的方式给出。过程调用指令和返回指令对状态标志位都没有影响。

1) 段内直接调用

指令格式及操作：

CALL nearproc ；SP←(SP)－2，(SP)＋1：(SP)←(IP)，IP＝(IP)＋disp

指令的操作数是一个近过程，该过程在本段内。指令汇编以后，得到 CALL 的下一条指令与被调用的过程入口地址的 16 位相对位移量 disp。指令操作是将指令指针 IP 的内容压入堆栈，然后将相对位移量 disp 加到 IP 上，使控制转到调用的过程。16 位相对位移量 disp 占 2 个字节，段内直接调用指令共有 3 个字节。

2) 段内间接调用

指令格式及操作：

CALL reg16/mem16 ；SP ←(SP)－2，(SP)＋1：(SP)←(IP)，IP←reg16/mem16

指令的操作数是 16 位的寄存器或存储器，其内容是一个近过程入口地址，指令操作是将指令指针 IP 的内容压入堆栈，然后将寄存器或存储器的内容送到 IP 中。

3) 段间直接调用

指令格式及操作：

CALL farproc ；SP←(SP)－2，(SP)＋1：(SP)←(CS)，CS←SEG far-proc

；SP←(SP)-2，(SP)＋1：(SP)←(IP)，IP← OFFSET far-proc

指令的操作数是一个远过程，该过程在另外的代码段内。段间直接调用指令先将 CS 中的段基值压入堆栈，并将远过程所在的段基值送 CS，再将 IP 中的偏移地址压入堆栈，然后将远过程的偏移地址 OFFSET far-proc 送 IP。

4) 段间间接调用

指令格式及操作：

CALL mem32 ；SP←(SP)－2，(SP)＋1：(SP)←(CS)，CS←mem32＋2

；SP←(SP)－2，(SP)＋1：(SP)←(IP)，IP←mem32

指令的操作数是 32 位的存储器地址，指令的操作是先将 CS 寄存器压入堆栈，并将存储器的后两个字节送 CS，再将 IP 中的偏移地址压入堆栈，然后将存储器的前两个字节送 IP，控制转到另一个代码段的远过程。

4. 过程返回指令

指令格式及操作：

1) 从近过程返回

RET；IP←((SP)＋1：(SP))，SP←(SP)＋2

RET popvalue ；IP←((SP)＋1：(SP))，SP←(SP)＋2，SP←(SP)＋popvalue

2) 从远过程返回

RET ；IP←((SP)＋1：(SP))，SP←(SP)＋2，CS←((SP)＋1：(SP))，SP←(SP)＋2

RET popvalue ；IP←((SP)＋1：(SP))，SP←(SP)＋2

；CS←((SP)＋1：(SP))，SP←(SP)＋2，SP←(SP)＋popvalue

过程体中总包含返回指令 RET，它将堆栈中的断点弹出，控制程序返回到原来调用过程的地方。通常，RET 指令的类型是隐含的，它自动与过程定义时的类型相匹配。但采用间接调用时，必须保证 CALL 指令类型与 RET 指令的类型相匹配，以免发生错误。此外，RET 指令可以带一个弹出值(popvalue)，这是一个 16 位的立即数，通常是偶数。弹出值表示返回时从堆栈舍弃的字节数。例如 RET 4，返回时从堆栈舍弃 4 个字节数。这些字节一般是调用前通过堆栈向过程传递的参数。

5．中断指令

在程序运行期间，有时会产生一些随机事件，如运算产生溢出、请求数据传送等；另外，在计算机的运行过程中，也可能会发生如输入／输出设备故障、存储器校验出错等意外情况。这些都会使得计算机不得不暂时中止正在运行的程序，而转去执行一组相应的处理程序，处理完毕后又返回原被中止的程序并继续执行，这样一个过程称为中断。

8086 中断系统分为外部中断(或叫硬件中断)和内部中断(或叫软件中断)。外部中断主要用来处理外设和 CPU 之间的通信，内部中断主要指中断指令引起的中断。

中断指令用于产生软件中断，以执行一段特殊的中断处理过程。中断指令主要有以下几个用途：

① 系统功能调用　为了便于各种程序设计，操作系统为用户提供了一些通用的功能子程序，如控制台输入／输出、文件读／写、软硬件资源管理、通信等。用户程序可以通过中断指令(如 INT21H)直接调用这些服务子程序，而不用再自己编写，从而大大简化了应用软件的开发；

② 用于实现一些特殊功能　如调试程序时单步运行、断点等；

③ 调用 BIOS 提供的硬件低层服务。

中断服务程序的入口地址通常被称为中断向量(Interrupt Vector)或中断矢量。8086 可处理 256 类中断，类型号为 0～255。每类中断有一个入口地址，需用 4 个字节存储 CS 和 IP，256 类中断的人口地址要占用 1K 字节，它们位于内存 00000～003FFH 的区域中。存储这些地址的连续空间称为中断向量表或中断矢量表，如图 5.36 所示。

图 5. 36　8086 中断向量表

1) 软中断产生指令

指令格式：

INT n　；n 为中断向量码(也称中断类型码)，取值范围为 0～255。指令执行时，CPU 根据 n 的值计算出中断向量的地址，然后从该地址中取出中断服务程序的入口地址，并转到该中断服务子程序去执行。中断向量地址的计算方法是将中断向量码 n 乘 4。INT 指令的具体操作步骤如下：

①　把标志寄存器的内容压入堆栈：SP←SP-2，(SP+1:SP)←FLAGS。

② 清除 IF 和 TF，以保证在中断服务子程序中不会被再次中断，并且也不会响应单步中断：TF←0，IF←0。

③ 把断点地址(INT 指令的下一条指令地址)的段地址和偏移地址压入堆栈：SP←SP-2，(SP+1:SP)←CS；SP←SP-2，(SP+1:SP)←IP。

④ 由 n×4 得到中断向量地址，并进而得到中断处理子程序的入口地址：IP←(n×4+1: n×4)；CS←(n×4+3: n×4+2)。

上述操作完成后，CS:IP 就指向中断服务程序的第一条指令，此后 CPU 开始执行中断服务子程序。INT n 指令除复位 IF 和 TF 外，对其他标志无影响。

从 CPU 执行中断指令的过程可以看出，INT 指令的基本操作与存储器寻址的段间间接调用指令非常相像，所不同的是：

INT 指令除保存断点外还要将标志寄存器 FLAGS 压入堆栈保存，而 CALL 指令则不必；

INT 影响 IF 和 TF 标志，而 CALL 指令不影响；

中断服务程序入口地址放在内存的固定位置，以便通过中断向量号找到它。而 CALL 指令可任意指定子程序人口地址的存放位置。

2) 溢出中断指令

INTO

有符号数运算中的溢出是一种错误，在程序中应尽量避免(如果避免不了，也希望能及时发现，否则程序再往下运行，其结果便毫无意义了)，为此 8086 指令系统专门提供了一条溢出中断指令，用来判断有符号数加减运算是否溢出。一般使 INTO 指令紧跟在有符号数加、减运算指令的后面。若运算使 OF=1，则 INTO 指令会使控制自动转到溢出中断处理程序，进行溢出处理。若 OF=0，则 INTO 指令不执行任何操作，使程序继续执行下一条指令。INTO 指令从检查到 OF=1 到转移至溢出中断处理程序的执行过程与 INT n 指令是类似的，只不过 INTO 指令是 n=4 的 INT 指令，其向量地址为 0010H。即 INTO 指令与 INT 4 指令调用的是同一个中断服务程序。

3)中断返回指令

IRET

中断返回指令 IRET 用于从中断服务子程序返回到被中断处继续执行原程序。任何中断服务子程序无论是由外部中断引起的，还是内部中断引起的，其最后一条指令都是 IRET 指令。该指令首先将堆栈中的断点地址弹出到 IP 和 CS，接着将 INT 指令执行时压入堆栈的标志字弹出到标志寄存器，以恢复中断前的标志状态。显然本指令对各标志位均有影响。

指令的操作为：

IP ←(SP+1:SP)，SP←SP+2；

CS ←(SP+1:SP)，SP←SP+2；

FLAGS←(SP+1:SP)，SP←SP+2。

六、字符串操作指令

8086 指令系统中有一组十分有用的串操作指令，这些指令的操作对象不只是单个字节或字，而是内存中地址连续的字节串或字串。在每次基本操作后，能够自动修改地址，为下一次操作做好准备。串操作指令还可以加上重复前缀;此时指令规定的操作将一直重复下去，直到完成预订的重复次数。

串操作指令共有以下五条：串传送指令(MOVS)、串装入指令(LODS)、串送存指令(STOS)、串比较指令(CMPS)和串扫描指令(SCAS)。

上述串操作指令的基本操作各不相同，但都具有以下几个共同特点：

(1) 用 SI 寻址源操作数，用 DI 寻址目标操作数，源操作数在数据段，隐含段寄存器 DS，可以段超越，目标操作数在附加段，隐含段寄存器 ES，不允许段超越。

(2) 每一次操作以后修改地址指针，是增量还是减量决定于方向标志 DF。当(DF)＝0 时，地址指针增量，即字节操作时地址指针加 1，字操作时地址指针加 2。当(DF)＝1 时，地址指针减量，即字节操作时地址指针减 1，字操作是地址指针减 2。

(3) 有的串操作指令可加重复前缀 REP，则指令规定的操作重复进行，重复操作的次数由 CX 寄存器决定。如果在串操作指令前加上重复前缀 REP，则 CPU 按以下步骤执行：

① 首先检查 CX 寄存器，若(CX)＝0，则退出重复串操作指令。

② 指令执行一次字符串基本操作。

③ 根据 DF 标志修改地址指针。

④ CX 减 1(但不改变标志)。

⑤ 转至下一次循环，重复以上步骤。

(4) 串操作指令的基本操作影响 ZF (如 CMPS、SCAS)可加重复前缀 REPE/REPZ 或 REPNE/REPNZ，此时操作重复进行的条件不仅要求(CX)≠0，而且同时要求 ZF 的值满足重复前缀中的规定(REPE 要求(ZF)＝1，REPNE 要求(ZF)＝0)

(5) 串操作汇编指令的格式可以写上操作数，也可以只在指令助记符后加字母“B”(字节操作)或“W”(字操作)，指令助记符后不加任何操作数。

1. 串传送指令

指令格式：

[REP]MOVS[ES:] dststring，[seg:]srcstring

[REP]MOVSB

[REP]MOVSW

MOVS 指令也称为字符串传送指令，它将一个字节或字从存储器的某个区域传送到另一个区域，然后根据方向标志 DF 自动修改地址指针。其执行的操作为：

(1) (ES)：(DI)←((DS)：(SI))

(2) SI←(SI)±1，DI←(DI)±1 (字节操作)或

SI←(SI)±2，DI←(DI)±2 (字操作)

其中，当方向标志 DF＝0 时用“＋”，当方向标志 DF＝1 时用“－”。串传送指令不影响状态标志寄存器。

以上内容均是任选项，即这些项可有可无。例如重复前缀 REP，可以加在串操作指令之前，也可以不加。

在第一种格式中，串操作指令给出源操作数和目标操作数，此时指令执行字节操作还是字操作，决定于这两个操作数定义时的类型。列出源操作数和目标操作数的作用有二。

首先，用以说明操作对象的大小(字节或字)，其次，明确指出涉及的段寄存器(seg)。指令执行时，实际仍用 SI 和 DI 寄存器寻址操作数。如果在指令中采用 SI 和 DI 来表示操作数，则必须用类型运算符 PTR 说明操作对象的类型。第一种格式的一个重要优点是可以对源字符串进行段重设(目的字符串的段基值只能在 ES，不可进行段重设)。

在第二种和第三种格式中，串操作指令字符的后面加上一个字母“B”或“W”，指出操作对象是字节串或字串。但要注意，在这两种情况下，指令后面不需要出现操作数。例如：

REP MOVS DATA2，DATAl ；操作数类型应预先定义

MOVS BUFFER2，ES：BUFFERl ；源操作数进行段重设

REP MOVS WORD PTR[DI]，[SI] ；用变址寄存器表示操作数

REP MOVSB ；字节串传送

MOVSW ； 字串传送

但以下表示方法是非法的：

MOVSB det，dfg

串传送指令常常与重复前缀联合使用，这样不仅可以简化程序，而且提高了运行速度，但必须把重复操作的次数(字节串长度)送 CX。例如下述程序片段用于传送 200 个字节：

LEA SI，BUFFERl ； SI←源串首地址指针

LEA DI，BUFFER2 ； DI←目的串首地址指针

MOV CX，200 ； CX←字节串长度

CLD ； 清方向标志 DF

REP MOVSB ； 传送 200 个字节

HLT

2. 串装入指令

指令格式：

LODS [seg：] srcstring

LODSB

LODSW

LODS 指令将字符串中的字节或字逐个装入累加器 AL 或 AX，其执行的操作为：

(1) AL←((DS)：(SI))或 AX←((DS)：(SI))

(2) SI←(SI)±1 (字节操作)或

SI←(SI)±2 (字操作)

其中，当方向标志 DF＝0 时用“＋”，当方向标志 DF＝1 时用“－”。串装入指令不影响状态标志寄存器。不加重复前缀。

将内存中 10 个非压缩的 BCD 码，按顺序送显示器显示。程序如下：

LEA SI，BUFFER ； SI←缓冲区首址

MOV CX，10 ； CX←字符串长度

CLD ； 清状态标志位 DF

MOV AH，02H ； AH←功能号

SC： LODSB ； 取一个 BCD 码到 AL

OR AL，30H ； BCD 码转换为 ASCII 码

MOV DL，AL ； DL←字符

INT 21H ；显示

DEC CX ；CX←(CX)－1

JNZ SC ；未完成 10 个字符则重复

HLT

3. 串送存指令

指令格式：

[REP]STOS [ES：]dst-string

[REP]STOSB

[REP]STOSW

STOS 指令是将累加器 AL 或 AX 的值送存到内存缓冲区的某个位置上。指令的基本操作为：

(1) (ES)：(DI)←(AL)或(ES)：(DI)←(AX)

(2) DI←(DI)±1 (字节操作)或

DI←(DI)±2 (字操作)

STOS 指令对状态标志位没有影响。指令若加上重复前缀 REP，则操作将一直重复进行，直到(CX)＝0。

将字符“##”装入以 AREA 为首址的 100 个字节中，程序如下：

LEA DI，AREA

MOV AX，‘##’

MOV CX，100

CLD

REP STOSW

HLT

4. 串比较指令(CMPS)

指令格式：

[REPE/REPNE] CMPS [seg：]src-string ，[ES：]dst-string

[REPE/REPNE] CMPSB

[REPE/REPNE] CMPSW

指令的基本操作为：

(1) ((DS)：(SI))－((ES)：(DI))

(2) SI←(SI)±1，DI←(DI)±1 (字节操作)或

SI←(SI)±2，DI←(DI)±2 (字操作)

CMPS 指令与其他指令有所不同，指令中的源操作数在前，而目标操作数在后。另外，CMPS 指令可以加重复前缀 REPE(也可以写成 REPZ)或 REPNE(也可以写成 REPNZ)，这是由于 CMPS 指令影响标志 ZF。如果两个被比较的字节或字相等，则(ZF)＝1，否则(ZF)＝0，REPE 或 REPZ 表示当(CX)≠0，且(ZF)＝1 时继续进行比较。REPNE 或 REPNZ 表示当(CX)≠0，且(ZF)＝0 时继续进行比较。

如果想在两个字符串中寻找第一个不相等的字符，则应使用重复前缀 REPE 或 REPZ，当遇到第一个不相等的字符时，就停止进行比较。但此时地址已被修改，即(DS：SI)和(ES：DI)已经指向下一个字节或字地址。应将 SI 和 DI 进行修正，使之指向所要寻找的不相等字符。同理，如果想要寻找两个字符串中第一个相等的字符，则应使用重复前缀 REPNE 或 REPNZ。但是也有可能将整个字符串比较完毕，仍未出现规定的条件(例如两个字符相等或不相等)，不过此时寄存器(CX)＝0，故可用条件转移指令 JCXZ 进行处理。

比较两个字符串，找出其中第一个不相等字符的地址。如果两个字符全部相同，则转到 MAT 进行处理。这两个字符串长度均为 20，首地址分别为 STl 和 ST2。程序如下：

```
LEA SI，ST1 ； SI←字符串 1 首地址
LEA DI，ST2 ； DI←字符串 2 首地址
MOV CX，20 ； CX←字符串长度
CLD ； 清方向标志 DF
REPE CMPSB ； 如相等，重复进行比较
JCXZ WE ； 转到 WE
DEC SI
DEC DI
HLT
WE ： MOV SI，0
MOV DI，0
HLT
```

5. 串扫描指令

指令格式：

```
[REPE/REPNE]SCAS[ES：]dst-string
[REPE/REPNE]SCASB
[REPE/REPNE]SCASW
```

SCAS 指令是在一个字符串中搜索特定的关键字，字符串的起始地址只能放在(ES：DI)

中，且不可以段超越。待搜索特定的关键字一定要放在 AL 或 AX 中。

SCAS 指令的基本操作为：

(1) (AL)－((ES)：(DI))或(AX)－((ES)：(DI))

(2) DI←(DI)±1 (字节操作)或

DI←(DI)±2 (字操作)

SCAS 指令将累加器的内容与字符串中的元素逐个进行比较，比较结果也反映在状态标志位上。SCAS 指令将影响状态标志位 SF、ZF、AF、PF、CF 和 OF。如果累加器的内容与字符串中的元素相等，则比较之后(ZF)＝1，因此，指令可以加上重复前缀 REPE 或 REPNE。前缀 REPE(即 REPZ)表示当(CX)≠0，且(ZF)＝1 时继续进行扫描。而 REPNE(即 REPNZ)表示当(CX)≠(0；且(ZF)＝0 时继续进行扫描。

在包含 100 个字符的字符串中，寻找第一个回车符 CR(其 ASCII 码为 0DH)，找到后将其地址保留在(DS：DI)中，并在屏幕上显示字符“Y”；如果字符串中没有回车符，则在屏幕上显示字符“N”。该字符串的首地址位 STRING。根据要求可编程如下：

```
LEA DI，STRING ；DI←字符串首址
MOV AL，0DH ；AL←回车符
MOV CX，100 ；CX←字符串长度
CLD ；清状态标志位 DF
REPNE SCASB ；如未找到，重复扫描
JZ MATCH ；如找到转 MATCH
MOV DL，N ；字符串中无回车，则 DL←“N”
JMP DSPY ；转到 DSPY
MATCH ：DEC DI ；DI←(D1)－1
MOV DL，'Y' ；DL←“Y”
DSPY： MOV AH，02H
INT 21H ；显示字符
HLT
```

在以上五条串操作指令中，有的指令有两个操作数，有的指令只有一个操作数。只有一个操作数时，有的指令是源操作数，有的指令是目标操作数。

关于串操作指令的重复前缀、操作数以及地址指针所用的寄存器等情况，如表 5.18 所示。

表 5.18 串操作指令的重复前缀、操作数和地址指针

指令	重复前缀	操作数	地址指针寄存器
MOVS	REP	目的，源	ES：DI，DS：SI
LODS	无源	DS：SI	
STOS	REP	目的	ES：DI
CMPS	REPE/REPNE	源，目的	DS：SI，ES：DI
SCAS	REPE/REPNE	目的	ES：DI

七、处理器控制指令

该类指令用于对 CPU 进行控制，例如对 CPU 中某些状态标志位的状态进行操作，以及使 CPU 暂停、等待等。8086/8088 指令系统的处理器控制指令可分为三组。

1. 标志位操作指令

1) CLC

清进位标志。指令的操作为 CF←0

2) STC

置进位标志。指令的操作为 CF←1

3) CMC

对进位标志求反。指令的操作为 CF← (CF)

4) CLD

清方向标志。指令的操作为 DF←0

5) STD

置方向标志。指令的操作为 DF←1

6) CLI

清中断允许标志。指令的操作为 IF←0

7) STI

置中断允许标志。指令的操作为 IF←1。在执行这条指令后，CPU 将允许外部的可屏蔽中断请求。

这些指令仅对有关状态标志位执行操作，而对其他状态标志位则没有影响。

2. 外部同步指令

1) HLT

执行 HLT 指令后，CPU 进入暂停状态。外部中断(当(IF)＝1)时的可屏蔽中断请求 INTR，

或非屏蔽中断请求 NMI 或复位信号 RESET 可使 CPU 退出暂停状态。HLT 指令对状态标志位没有影响。

2) WAIT

如果 8086/8088 CPU 的 TEST 引脚上的信号无效(即高电平)，则 WAIT 指令使 CPU 进入等待状态。一个被允许的外部中断或 TEST 信号有效，可使 CPU 退出等待状态。在允许中断的情况下，一个外部中断请求将使 CPU 离开等待状态，转向中断服务程序。此时被推入堆栈进行保护的断点地址即是 WAIT 指令的地址，因此从中断返回后，又执行 WAIT 指令，CPU 再次进入等待状态。

如果 TEST 信号变低(有效)，则 CPU 不再处于等待状态，开始执行下面的指令。但是，在执行完下一条指令之前，不允许有外部中断。

本指令对状态标志位没有影响。WAIT 指令的用途是使 CPU 本身与外部的硬件同步工作。

3) ESC

指令格式

ESC extop，src

ESC 指令使其他处理器可使用 8086/8088 的寻址方式，并从 8086/8088 CPU 的指令队列中取得指令。以上指令格式中的 extop 是其他处理器的一个操作码(外操作码)，src 是一个存储器操作数。执行 ESC 指令时，8086/8088 CPU 访问一个存储器操作数，并将其放在数据总线上，供其他处理器使用。此外没有其他操作。例如：协处理器 8087 的所有指令机器码的高五位都是“11011”，而 8086/8088 的 ESC 指令机器码的第一个字节恰是“1101lXXX”，因此，对于这样的指令，8086/8088 CPU 将其视为 ESC 指令，它将存储器操作数置于总线上，然后由 8087 来执行该指令，并使用总线上的操作数。8087 的指令系统请参考有关资料。

ESC 指令对状态标志位没有影响。

4) LOCK

这是一个特殊的可以放在任何指令前面的单字节前缀。这个指令前缀迫使 8086CPU 的总线锁定信号线 LOCK 维持低电平(有效)，直到执行完下一条指令。外部硬件可接收这个 LOCK 信号。在其有效期间，禁止其他处理器对总线进行访问。共享资源的多处理器系统中，必须提供一些手段对这些资源的存取进行控制，指令前缀 LOCK 就是一种手段。

3. 空操作指令 NOP

执行 NOP 指令时不进行任何操作，但占用 3 个时钟周期，然后继续执行下一条指令。NOP 指令对状态标志位没有影响，指令没有操作数。

5.6. 8086CPU 的存储器扩展

5.6.1. 存储器扩展概述

存储器芯片的外部引脚按功能分为数据线（DB）、地址线（AB）和控制线（CB）。在 CPU 对存储器进行读写操作时，首先在地址总线上给出地址信号，然后发出相应的读或写控制信号，最后才能在数据总线上进行数据交换，所以 CPU 与存储器的连接包括地址线、数据线和控制线的连接等三个部分。在连接时要考虑以下几个问题：

(1) CPU 总线的负载能力

一般来说，CPU 总线的直流负载能力可带一个 TTL 负载，目前存储器基本上是 MOS 电路，直流负载很小，主要负载是电容负载。因此在小型系统中，CPU 可以直接和存储器芯片相连，在较大的系统中，考虑到 CPU 的驱动能力，必要时应加上数据缓冲器(例如 74LS245)或总线驱动器来驱动存储器负载。

(2) CPU 的时序和存储器存取速度之间的配合

CPU 在取指令和读/写操作数时，有它自己固定的时序，应考虑选择何种存储器来与 CPU 时序相配合。若存储器芯片已经确定，应考虑如何实现等待周期的插入。

(3) 存储器的地址分配和片选

内存分为 ROM 区和 RAM 区，RAM 又分为系统区和用户区，每个芯片的片内地址，由 CPU 的低位地址来选择。一个存储器系统有多片芯片组成，片选信号由 CPU 的高位地址译码后敢得。应考虑采用何种译码方式，实现存储器的芯片选择。

(4) 控制信号的连接

8086 CPU 与存储器交换信息时，提供了以下几个控制信号；M/$\overline{IO}$、$\overline{RD}$、$\overline{WR}$、ALE、READY、DT/$\overline{R}$ 和 $\overline{DEN}$ 等，这些信号与存储器要求的控制信号如何连接才能实现所需要的控制功能。

5.6.2. CPU 与存储器地址线的连接

一个存储器系统通常由许多存储器芯片组成，对存储器的寻址必须有两个部分，通常是将低位地址线连到所有存储器芯片，实现片内寻址，将高位地址线通过译码器或线性组合后输出作为芯片的片选信号，实现片间寻址。地址线的连接决定了存储器的地址分配，常见的有三种实现存储器地址选择的方法。

(1) 线性选择法

这种方法直接用 CPU 地址总线中某一高位线作为存储器芯片的片选信号，简称为线选法。线选法的优点是连接简单，片选信号的产生不需要复杂的逻辑电路，只用一条地址线与 MREQ 的简单组合就可产生有效的片选信号 $\overline{CS}$。

当采用线选法时，若低位地址线用于字选，高位地址线用作线选，当高位地址未全部用完、而又没有对其控制时，会出现地址的不连续性和多义性，这是线选法的两大缺点。线选法还有另一种局限：即使所有高位地址线都用作线选，其能寻址的存储空间十分有限。在大系统中，线选法有限的寻址能力限制了存储器系统的扩展，这也是它的一个弱点。因此，为避免地址的不连续性和多义性、加强系统存储器的扩展能力，一般另一种寻址方法--全译码法。

(2) 全译码法

全译码法将高位地址线全部作为译码器的输入，用译码器的输出作片选信号。在这种寻址方法中，低位地址线用作字选，与芯片的地址输入端直接相连；高位地址线统统连接进译码电路，用来生成片选信号。这样，所有的地址线均参与片内或片外的地址译码，不会产生地址的多义性和不连续性。在全译码方式中，译码电路的核心常用一块译码器充当，如 3-8 译码器 74LS138 等。

(3) 部分译码法

部分译码选择方式是将高位地址线中的几位经过译码后作为片选控制，它是线性选择与全译码选择法的混合方式。

RAM 芯片 Intel 6264 容量为 8K×8 位，用 2 片静态 RAM 芯片 6264，组成 16K×8 位的存储器系统。地址选择的方式是将地址总线低 13 位(A12～A0)并行地与存储器芯片的地址线相连，而 $\overline{CS}$ 端与高位地址线相连。

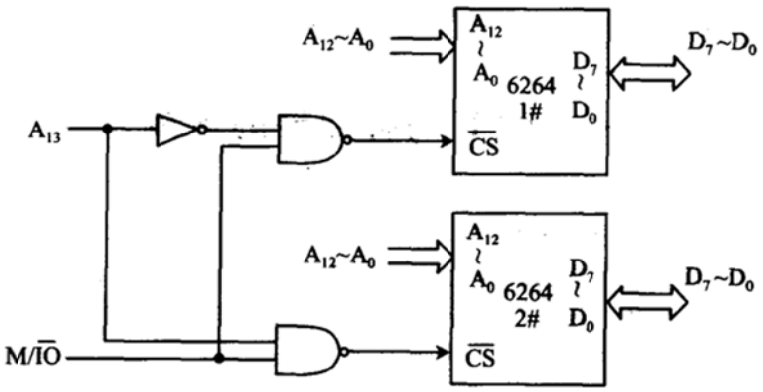


图 5. 37　线性选择

为了区分 6264 两组不同的芯片，可用地址线 A13～A19 中任一根地址线来控制，如图 5.37 所示，用 A13 来控制。A13=0 则选中 1#芯片，A13=1 则选中 2#芯片，此时 1#芯片的段内地址为 0000～lFFFH，2#芯片的地址为 2000H～3FFFH。但是只要 A13=0， A14～A19 为任意值都选中 1#芯片，而只要 A13=1，A14～A19 为任意值都选中 2#芯片，所以它们的地址是重叠的。在一个段 64KB 中，地址重叠区有 4 个，即有 4 组地址可用于 1#号芯片寻址：

0000～1FFFH，4000H～5FFFH，8000H～9FFFH，C000H～DFFFH

4 组地址可用于 2#芯片寻址：

2000H～3FFFH，6000H～7FFFH，A000H～BFFFH， E000H～FFFFH

至于不同的段内重叠区就更多了，这种寻址方式称为线选方式。

如图 5.38 所示为译码电路，其中片选信号线 1#~7#接存储器，试确定在一个段（64k）中的存储器地址范围。

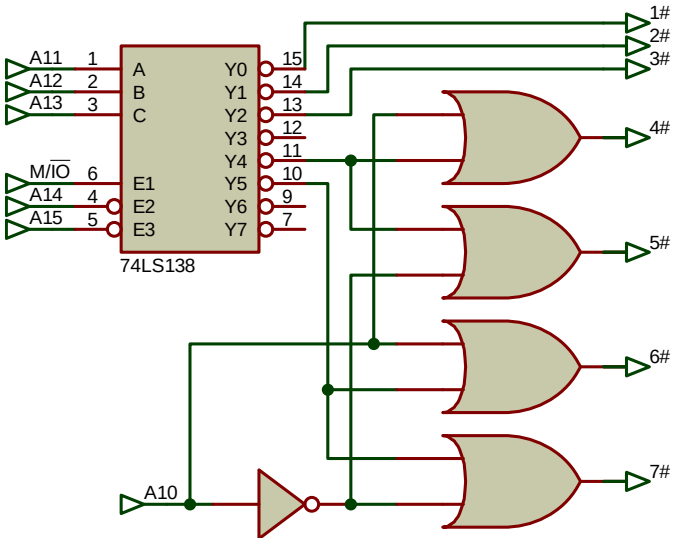


图 5. 38　译码电路

本电路采用 3-8 译码器 74LS138，该芯片的真值表见表所示。由电路可知，地址线 A15～A10 参与片选译码，它们与片选信号线 1#~7#的逻辑关系见表，表中同时给出了 1#~7#接存储器的地址范围。

E1　E2　E3	C　B　A	输出
1　0　0	0　0　0	Y0=0，其余为 1
1　0　0	0　0　1	Y1=0，其余为 1
1　0　0	0　1　0	Y2=0，其余为 1
1　0　0	0　1　1	Y3=0，其余为 1
1　0　0	1　0　0	Y4=0，其余为 1
1　0　0	1　0　1	Y5=0，其余为 1
1　0　0	1　1　0	Y6=0，其余为 1
1　0　0	1　1　1	Y7=0，其余为 1

	A15 A14 A13 A12 A11 A10	输出
1#	0　0　0　0　0　×	0000H～07FFH
2#	0　0　0　0　1　×	0800H～0FFFH

3#	0 0 0 1 0 ×	1000H～17FFH
4#	0 0 1 0 0 0	2000H～23FFH
5#	0 0 1 0 0 1	2400H～27FFH
6#	0 0 1 0 1 0	2800H～2BFFH
7#	0 0 1 0 1 1	2C00H～2FFFH

5.6.3. CPU 与存储器数据线及控制线的连接

8086 CPU 有 20 位地址线，可寻址 1MB 的存储空间。8086 CPU 数据线有 16 位，可以读/写一个字节，也可读/写一个字。与 8086 CPU 相连的存储器是用 2 个 512KB 的存储体组成的，它们分别称为低位（偶地址）存储体和高位（奇地址）存储体，用 A0 和 $\overline{BHE}$ 信号分别来选择两个存储体，用 A19～A1 来选择存储体体内的地址。若 A0=0 选中偶地址存储体，它的数据线连到数据总线低 8 位 D7～D0，若 $\overline{BHE}$ =0 选中奇地址存储体，它的数据线连到数据总线高 8 位 D15～D8。若读写一个字，A0 和 $\overline{BHE}$ 均为 0，两个存储体全选中。

8086 CPU 与存储器芯片连接的控制信号主要有地址锁存信号 ALE，读选通信号 $\overline{RD}$，写选通信号 $\overline{WR}$，存储器或 I/O 选择信号 M/$\overline{IO}$，数据允许输出信号 $\overline{DEN}$，数据收发控制信号 DT/$\overline{R}$，准备好信号 READY。

要求用 4K×8 的 EPROM 芯片 2732, 8K×8 的 RAM 芯片 6264，译码器 74LS138 构成 8K 字 ROM 和 8K 字 RAM 的存储器系统，系统配置为最小模式。图 5.39 给出了系统连接图。图中未画出对 ROM 芯片数据线的低 8 位与高 8 位的选择，可将 A0 和 $\overline{BHE}$ 分别与 Y0 信号相“与”来实现此选择。

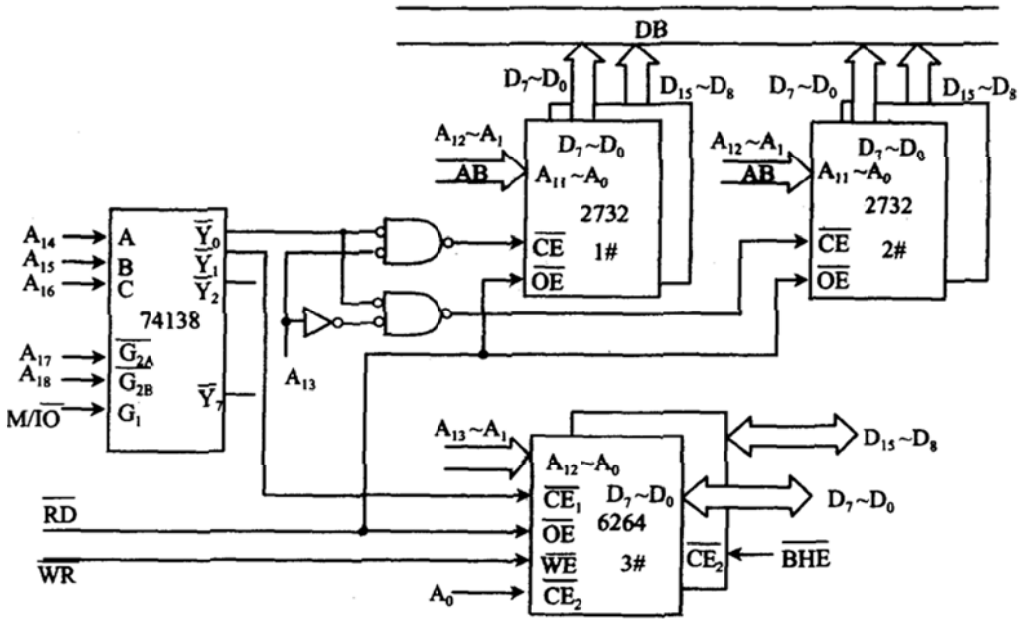


图 5. 39　一个存储子系统例

ROM 芯片，8K 字用 4 片 2732 芯片组成，片内用 12 根地址线 A1～A12 来寻址。RAM 芯片，8K 字用 2 片 6264 芯片组成，片内用 13 根地址线 A1～A13 寻址。芯片选择由 74LS138 译码器输出 Y0、Y1 完成。

ROM 芯片由 $\overline{RD}$ 信号(连 $\overline{OE}$ 端)来完成数据读出。RAM 芯片由 $\overline{RD}$ 信号(连 $\overline{OE}$ 端)和 $\overline{WR}$ 信号(连 $\overline{WE}$ 端)来完成数据读 / 写，A0 和 $\overline{BHE}$ 用来区分数据线的低 8 位及高 8 位。

由于 ROM 芯片容量撕 4KX8 位，RAM 芯片容量为 8K×8 位，用 A13 和 Y0 输出进行二次译码，来选择两组 ROM 芯片，这样可以保证存储器系统地址连续。

74LS138 译码器的输入端 C，B，A 分别连地址线 A16～A14，控制端 G1、$\overline{G_{2A}}$ 和 $\overline{G_{2B}}$ 分别连 M/$\overline{IO}$ 和 A17、A18，计算得到存储器的地址范围为：

1#芯片：00000H～01FFFH

2#芯片：02000H～03FFFH

3#芯片：04000H～07FFFH

5.7. 8086CPU 的中断系统

5.7.1. 中断概述

1. 中断的概念

中断是 CPU 在执行当前程序的过程中，当出现某些异常事件或某种外部请求时，使得

CPU 暂时停止正在执行的程序(即中断)，转去执行外围设备服务的程序。当外围设备服务的程序执行完后，CPU 再返回暂时停止正在执行的程序处(即断点)，继续执行原来的程序。这种中断就是人们通常所说的外部中断；但是随着计算机体系结构不断的更新换代和应用技术的日益提高，中断技术发展的速度也是非常迅速的，中断的概念也随之延伸，中断的应用范围也随之扩大。除了传统的外围部件引起的硬件中断外，又出现了内部的软件中断概念。

在 Pentium 中则更进一步丰富了软件中断的种类，延伸了中断的内涵。它把许多在执行指令过程中产生的错误也归并到了中断处理的范畴，并将它们和通常意义上的内部软件中断一起统称其为异常，而将传统的外部中断简称为中断。由此可见，中断和异常对于 Pentium 微处理机来说是有区别的，其主要差别在于：中断用来处理 CPU 以外的异常事件，而异常则是用来处理在执行指令期间，由 CPU 本身对检测出来的某些异常事情做出的响应。当再次执行产生异常的程序或数据时，这种异常总是可以再次出现。而由外围部件引起的硬件中断，一般说来与当前的执行程序无关；但是，当中断和异常在使微处理机暂时停止执行当前的程序，去执行更高优先级别的程序时，却是一样的。

外部中断和内部软件中断就构成了一个完整的中断系统。发出中断请求的来源非常多，不管是由外部事件而引起的外部中断，还是由软件执行过程而引发的内部软件中断，凡是能够提出中断请求的设备或异常故障，均被称为中断源。

2. 中断源

引起中断的原因或发出中断请求的来源，称为中断源。中断源有以下几种：

(1) 外设中断源。一般有键盘、打印机、磁盘、磁带等，工作中要求 CPU 为它服务时，会向 CPU 发送中断请求。

(2) 故障中断源。当系统出现某些故障时(如存储器出错、运算溢出等)，相关部件会向 CPU 发出中断请求，以便使 CPU 转去执行故障处理程序来解决故障。

(3) 软件中断源。在程序中向 CPU 发出中断指令(8086 为 INT 指令)，可迫使 CPU 转去执行某个特定的中断服务程序，而中断服务程序执行完后，CPU 又回到原程序中继续执行 INT 指令后面的指令。

(4) 为调试而设置的中断源。系统提供的单步中断和断点中断，可以使被调试程序在执行一条指令或执行到某个特定位置处时，自动产生中断，从而便于程序员检查中间结果，寻找错误所在。

3. 中断的优先级

在实际系统中，常常遇到多个中断源同时请求中断的情况，这时 CPU 必须确定首先为哪一个中断源服务，以及服务的次序。解决的方法是用中断优先排队的处理方法。即根据中断源要求的轻重缓急，排好中断处理的优先次序，即优先级(Priority)，又称优先权。先响应优先级最高的中断请求。有的微处理器有两条或更多的中断请求线，而且已经安排好中断的优先级，但有的微处理器只有一条中断请求线。凡是遇到中断源的数目多于 CPU 的中断请求线的情况时，就需要采取适当的方法来解决中断优先级的问题。

另外，当 CPU 正在处理中断时，也要能响应优先级更高的中断请求，而屏蔽掉同级或较低级的中断请求即所谓多重中断的问题。

通常，解决中断的优先级的方法有以下几种：

(1) 软件查询确定中断优先级

软件查询中断方式，是将各个外设的中断请求信号通过或门相“或”后，送到 CPU 的 INTR 端，同时把几个外设的中断请求状态位组成一个端口，赋以端口号，如图？所示。任一外设有中断请求，CPU 响应中断后进人中断处理子程序，用软件读取端口内容，逐位查询端口的每位状态，查到哪个外设有请求中断，就转入哪个外设的中断服务程序。查询程序的次序，决定了外设优先级别的高低，先测试的中断源优先级别最高。当然在软件查询程序中也可用移位或屏蔽法来改变端口各位的测试次序，但查询时间较长，对中断源较多的情况不合适。

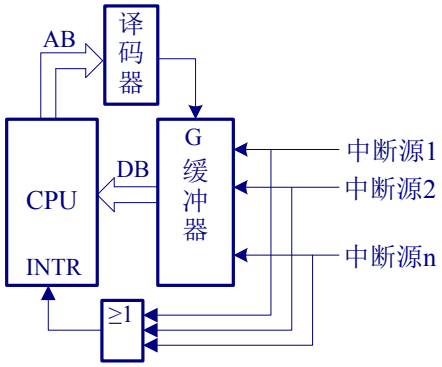


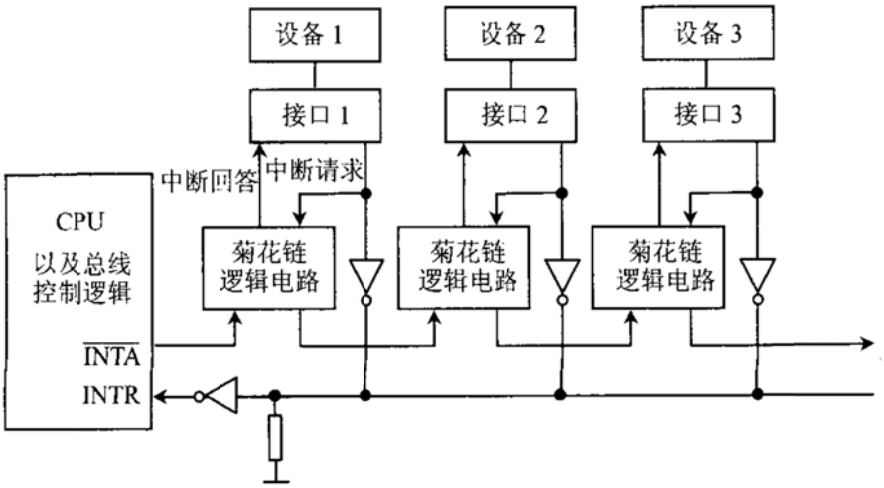
图 5. 40　软件查询中断方式的硬件接口框图

(2) 硬件查询确定优先级--菊花链法

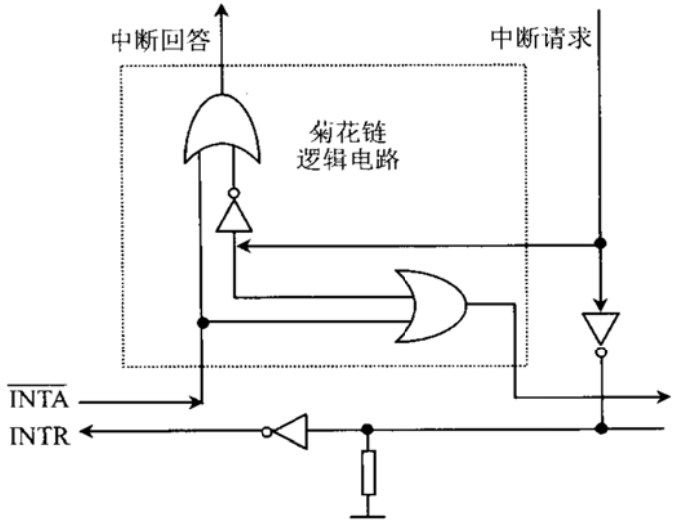
菊花链法是采用硬件查陶优先的方式，它是在每个外设的对应接口上连接一个逻辑电路构成一个链来控制中断响应信号的通路，图？给出了它的原理图。

当任一外部设备申请中断后，中断请求信号送到 CPU 的 INTR 端，CPU 发出 $\overline{\text{INTA}}$ 中断响应信号。当前一组的外设没有发出中断申请时，$\overline{\text{INTA}}$ 信号会沿着菊花链线路向后传递到发出中断请求的接口。当某一级的外设发出了中断申请，此级的逻辑电路就阻塞了 $\overline{\text{INTA}}$ 的通路，后面的外设接口不能接收到 $\overline{\text{INTA}}$ 信号。此级接口收到 $\overline{\text{INTA}}$ 信号后撤销中断请求信号，向总线发送中断类型号，从而 CPU 可以转入中断处理。

当多个外设接口同时申请中断时，显然最接近 CPU 的接口优先得到中断响应。菊花链的排列，使外设接口不会竞争中断响应信号 $\overline{\text{INTA}}$，从硬件线路上就决定了越靠近 CPU 的外设接口，优先级越高，其中断优先得到响应。



(a) 菊花链



(b) 菊花链逻辑电路

图 5. 41　菊花链优先查询法

(3) 中断优先级编码电路—矢量中断优先级

矢量中断优先级的设置是采用中断优先级控制器。图 5.42 给出了它的典型设计原理框图。

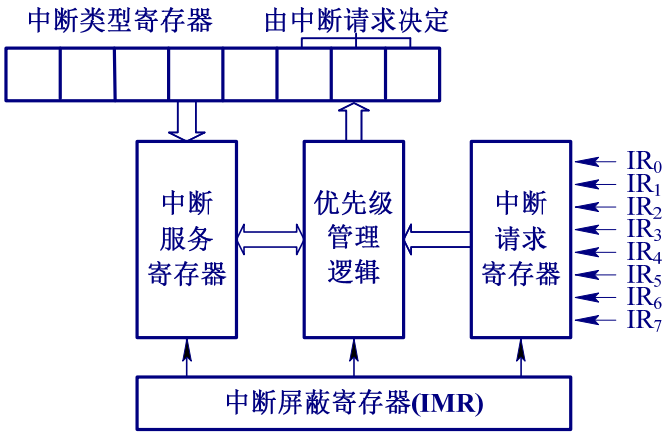


图 5. 42　矢量中断优先权控制器的原理图

外设可以有 8 个中断请求 IR0～IR7，送入中断请求寄存器，中断屏蔽寄存器可由用户设置屏蔽某几位的中断请求。中断优先级管理逻辑电路判别出最高优先级中断请求，将其中断级转换成 3 位码，送到中断类型寄存器的低 3 位及当前中断服务寄存器。此后，中断优先级控制器向 CPU 发出中断请求信号，当 CPU 开放中断时，CPU 发出中断响应信号，开始一个中断处理过程，中断处理结束引起中断服务寄存器对应位清 0，级别较低的中断请求才能得到响应。

可编程中断控制器 8259A 就是这样一种结构的芯片，通过软件的设置可以有多种优先

权设置方式。

4. 中断的嵌套

当 CPU 执行优先级较低的中断服务程序时，允许响应优先级比它高的中断源请求中断，而挂起正在处理的中断，这就是中断嵌套或称多重中断。此时，CPU 将暂时中断正在进行着的级别较低的中断服务程序，优先为级别高的中断服务，待优先级高的中断服务结束后，再返回到刚才被中断的较低优先级的那一级，继续为它进行中断服务。

多重中断的响应与单级中断的区别有以下几点：

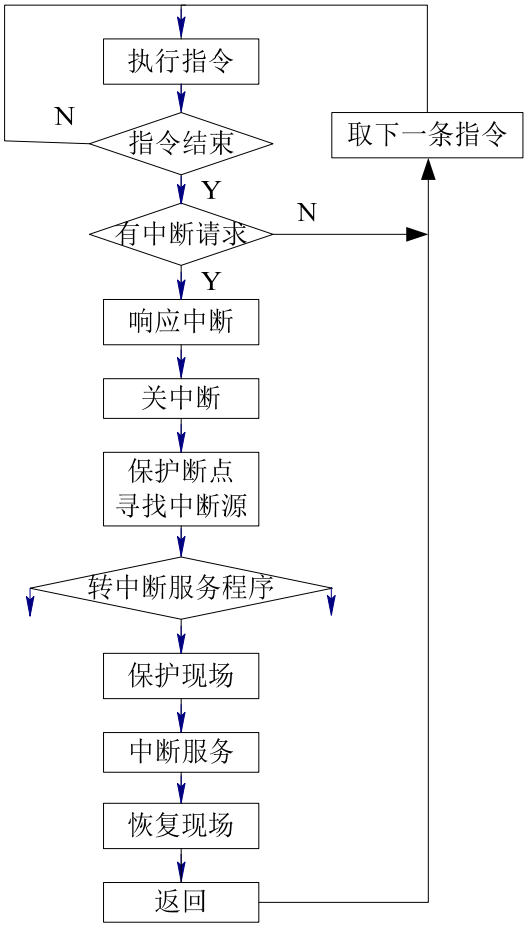
(1) 加入屏蔽本级和较低级中断请求的环节。这是为了防止在进行中断处理时，不致受到来自本级和较低级中断的干扰，并允许优先级比它高的中断源进行中断。

(2) 在进行中断服务之前，要开中断。因为如果中断仍然处于禁止状态，则将阻碍较高级中断的中断请求和响应，所以必须在保护现场、屏蔽本级及较低级中断完成之后，开中断，以便允许进行中断嵌套。

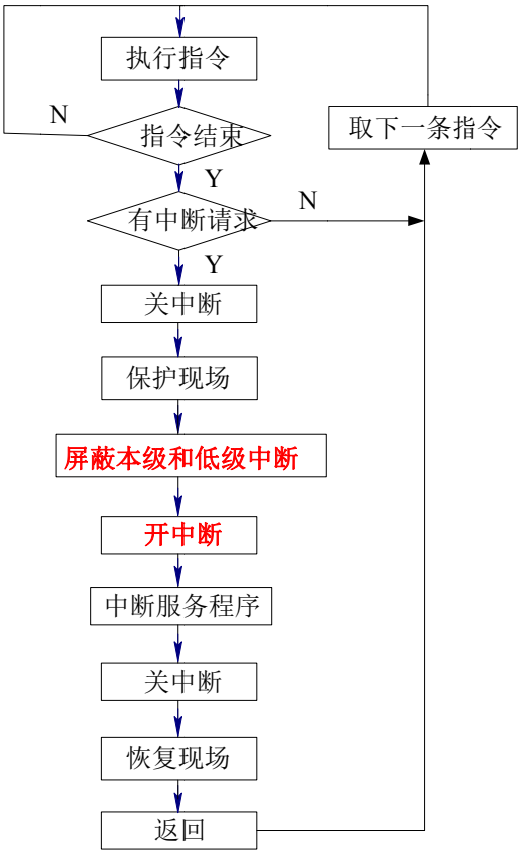
(3) 中断服务程序结束之后，为了使恢复现场过程不致受到任何中断请求的干扰，必须安排并执行关中断指令，将中断关闭，才能恢复现场。

(4) 恢复现场后，应该安排并执行开中断指令，重新开中断，以便允许任何其他等待着的中断请求有可能被 CPU 响应。应当指出，只有在执行了紧跟在开中断指令后面的一条指令以后，CPU 才重新开中断。一般紧跟在开中断指令后的是返回指令 RET，它将把原来被中断的服务程序的断点地址弹回 IP 及 CS，然后 CPU 才能开中断，响应新的中断请求。

单级中断和多级中断的响应流程见图 5.43。


执行指令
N
指令结束
取下一条指令
Y
有中断请求
N
Y
响应中断
关中断
保护断点
寻找中断源
转中断服务程序
保护现场
中断服务
恢复现场
返回

(a) 单级中断响应流程



(b) 单级中断响应流程

图 5. 43　中断响应流程

5.7.2. 8086 的中断系统

1. 中断类型

8086CPU 具有一个简单而灵活的中断系统，可处理 256 种不同的中断请求。8086 的中断源如图所示。根据中断源是来自 CPU 内部还是外部，通常将所有中断源分为两类：外部中断源和内部中断源，对应的中断称为外部中断或内部中断。

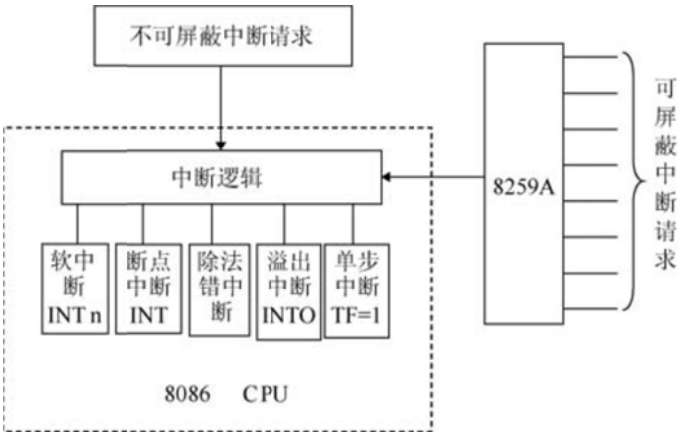


图 5. 44　8086 中断源

外部中断源即硬件中断源，来自 CPU 外部。8086 CPU 提供了两个引脚来接收外部中断源的中断请求信号：可屏蔽中断请求引脚和不可屏蔽中断请求引脚。

通过可屏蔽中断请求引脚输入的中断请求信号称作可屏蔽中断请求，对这种中断请求 CPU 可响应，也可不响应，具体取决于标志寄存器中 IF 标志位的状态。通过不可屏蔽中断请求引脚输入的中断请求信号称作不可屏蔽中断请求，这种中断请求 CPU 必须响应。

内部中断源是来自 CPU 内部的中断事件，这些事件都是特定事件，一旦发生，CPU 即调用预定的中断服务程序去处理。内部中断主要有以下几种情况：

1) 除法错误

当执行除法指令时，如果除数为 0 或商数超过了最大值，CPU 会自动产生类型为 0 的除法错误中断。

2) 软件中断

执行软件中断指令时，会产生软件中断。8086 系统中，设置了三条中断指令，分别是：

(1) 中断指令 INT n：用户可以用 INT n 指令来产生一个类型为 n 的中断，以便让 CPU 执行 n 号中断的中断服务程序。

(2) 断点中断 INT 3：执行断点指令 INT 3，将引起类型为 3 的断点中断，这是调试程序专用的中断。

(3) 溢出中断 INTO：如果标志寄存器中溢出标志位 OF 为 1，在执行了 INTO 指令后，产生类型为 4 的溢出中断。

3) 单步中断

当标志寄存器的标志位 TF 置 1 时，8086 CPU 处于单步工作方式。CPU 每执行完一条当标志寄存器的标志位 TF 置 1 时，8086 CPU 处于单步工作方式。CPU 每执行完一条指令，自动产生类型为 1 的单步中断，直到将 TF 置 0 为止。

单步中断和断点中断一般仅在调试程序时使用。调试程序通过为系统提供这两种中断的中断服务程序的方式，在发生断点或单步中断后获得 CPU 控制权，从而可以检查被调试程序(中断前 CPU 运行的程序)之状态。

为了解决多个中断同时申请时响应的先后顺序问题，系统将所有的中断划分为四级，以 0 级为最高，依次降低，各级情况如下：

(1) 0 级——除单步中断以外的所有内部中断。

(2) 1 级——不可屏蔽中断。

(3) 2 级——可屏蔽中断。

(4) 3 级——单步中断。

不同级别的中断同时申请时，CPU 根据级别高低依次决定响应顺序。

2. 中断类型号

由于系统中存在许多中断源，当中断发生时，CPU 就要进行中断源的判断。只有知道了中断源，CPU 才能调用相应的中断服务程序来为其服务。为了标记中断源，人们给系统中的每个中断源指定了一个唯一的编号，称为中断类型号。CPU 对中断源的识别就是获取当前中断源的中断类型号，在 8086 系统中的实现如下所述：

(1) 可屏蔽中断(硬件中断)：所有通过可屏蔽中断请求引脚向 CPU 发送的中断请求，都必须由中断控制器 8259A 管理。CPU 在准备响应其中断请求时，会给 8259A 发一个中断响应信号，8259A 收到这一信号后，会将发出中断申请外设的中断类型号通过系统数据总线发送给 CPU。

(2) 软件中断：在中断指令 INT n 中，参数 n 即为中断类型号。

(3) 除上面两种情况外，其余中断都是固定类型号，主要是内部中断：如除法错(类型 0)、单步中断(类型 1)、断点中断 INT 3(类型 3)、溢出中断 INTO(类型 4)等。外部中断中不可屏蔽中断的类型号也是固定的(类型 2)。

8086 系统中，中断类型号范围为 0～FFH，即最多有 256 个中断源。

3. 中断向量表

CPU 在响应中断时，要执行该中断源对应的中断服务程序，中断服务程序的入口地址由 CPU 通过查找中断向量表来得知。中断服务程序的地址叫做中断向量，将全部中断向量集中在一张表中，即中断向量表。中断向量表的位置固定在内存的最低 1K 字节中，即 00000H～003FFH 处。这张表中存放着所有中断服务程序的入口地址，而且根据中断类型号从小到大依次排列，每一个中断服务程序的入口地址在表中占 4 字节：前两个字节为偏移量，后两个字节为段基址。因系统中共有 256 个中断源，而每个中断服务程序入口地址又占 4 字节，故中断向量表共占 256×4＝1K 个字节，如图 9.1 所示。

那么，在系统中，实际上由谁来提供中断服务程序，并填写中断向量表中的内容呢？主要是 ROM BIOS 和 DOS。它们填写了中断向量表的大部分项目并提供了相应的中断服务程序。此外，主板上的各种硬件插卡(如果它们向系统提供中断服务)及在 DOS 下运行的以中断方式工作的内存驻留程序(如鼠标驱动程序、后台打印程序等)也会填写部分中断向量表项目并提供相应的中断服务程序。最后，还有部分中断向量表项目无人填写，也无人提供对应的中断服务程序，这部分中断是保留给用户用的。

图 5. 45　中断向量表结构

5.7.3. 8086 的中断处理过程

虽然不同类型的计算机系统中中断系统有所不同，但实现中断的过程是相同的。中断的处理过程一般有以下几步：中断请求、中断响应、中断处理、中断返回。8086CPU 的中断响应过程如图所示。

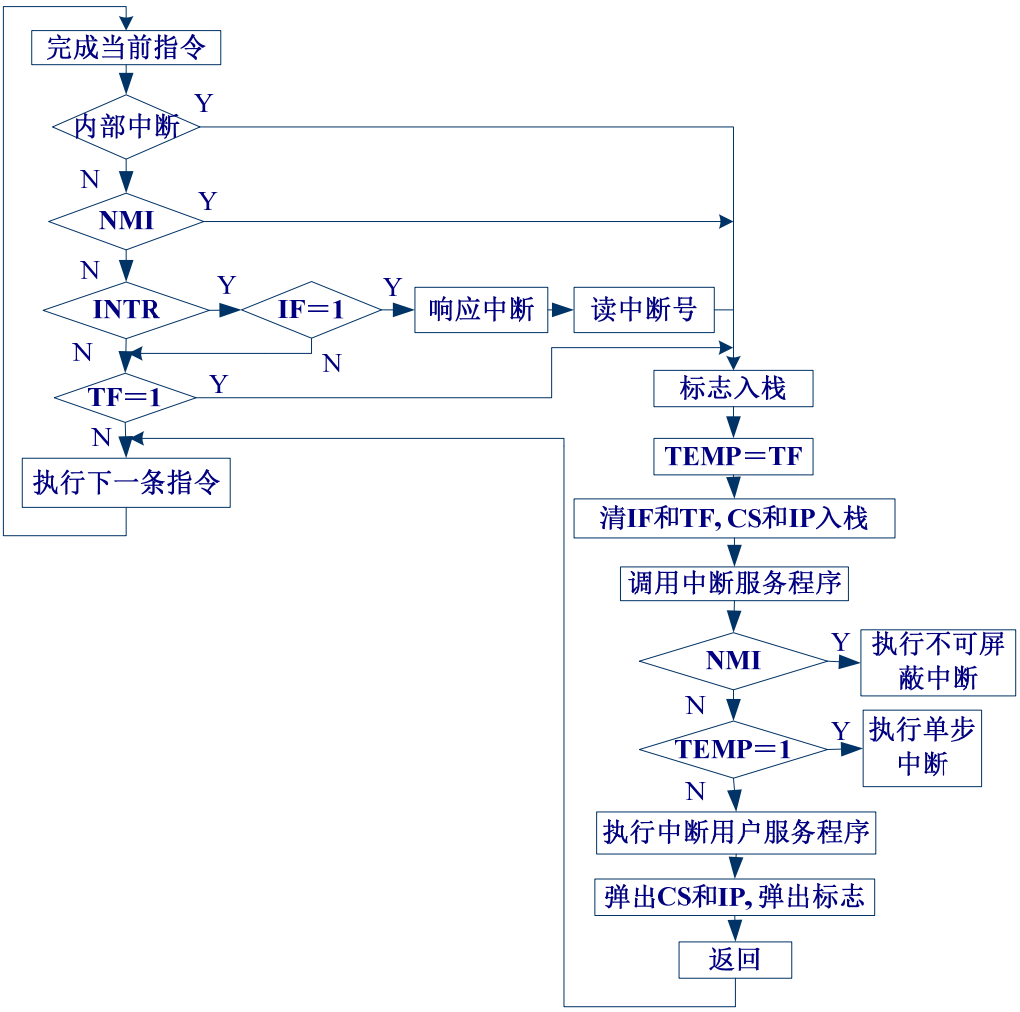


图 5. 46　8086CPU 的中断响应过程

1. 中断请求

中断处理的第一步是中断源发出中断请求，这一过程随中断源类型的不同而出现不同的特点，具体如下：

外部中断源的中断请求

当外部设备要求 CPU 为它服务时，需要发一个中断请求信号给 CPU 进行中断请求。8086 CPU 有两根外部中断请求引脚 INTR 和 NMI 供外设向其发送中断请求信号用，这两根引脚的区别在于 CPU 响应中断的条件不同。

CPU 在执行完每条指令后，都要检测中断请求输入引脚，看是否有外设的中断请求信号。根据优先级，CPU 先检查 NMI 引脚再检查 INTR 引脚。

INTR 引脚上的中断请求称为可屏蔽中断请求，CPU 是否响应这种请求取决于标志寄存器的 IF 标志位的值。IF＝1 为允许中断，CPU 可以响应 INTR 上的中断请求；IF＝0 为禁止中断，CPU 将不理会 INTR 上的中断请求。

由于外部中断源有很多，而 CPU 的可屏蔽中断请求引脚只有一根，这又产生了如何使得多个中断源合理共用一根中断请求引脚的问题。解决这个问题的方法是引入 8259A 中断

控制器，由它先对多路外部中断请求进行排队，根据预先设定的优先级决定在有中断请求冲突时，允许哪一个中断源向 CPU 发送中断请求。

NMI 引脚上的中断请求称为不可屏蔽中断请求(或非屏蔽中断请求)，这种中断请求 CPU 必须响应，它不能被 IF 标志位所禁止。不可屏蔽中断请求通常用于处理应急事件。在 PC 系列机中，RAM 奇偶校验错、I/O 通道校验错和协处理器 8087 运算错等都能够产生不可屏蔽中断请求。

内部中断源的中断请求

CPU 的中断源除了外部硬件中断源外，还有内部中断源。内部中断请求不需要使用 CPU 的引脚，它由 CPU 在下列两种情况下自动触发：其一是在系统运行程序时，内部某些特殊事件发生(如除数为 0，运算溢出或单步跟踪及断点设置等)；其二是 CPU 执行了软件中断指令 INT n。所有的内部中断都是不可屏蔽的，即 CPU 总是响应(不受 IF 限制)。

2. 中断响应

CPU 接受了中断源的中断请求后，便进入了中断处理的第二步：中断响应。这一过程也随中断源类型的不同而出现不同的特点，具体如下：

可屏蔽外部中断请求的中断响应

可屏蔽外部中断请求中断响应的特点是：(1) 由于外设(实际上是中断控制器 8259A，本处为求简单，统称为外设)不知道自己的中断请求能否被响应，所以 CPU 必须发信号(用 $\overline{\text{INTA}}$ 引脚)通知其中断请求已被响应。(2) 由于多个外设共用一根可屏蔽中断请求引脚，CPU 必须从中断控制器处取得中断请求外设的标识——中断类型号。

当 CPU 检测到外设有中断请求(即 INTR 为高电平)时，CPU 又处于允许中断状态，则 CPU 就进入中断响应周期。在中断响应周期中，CPU 自动完成如下操作：

(1) 连续发出两个中断响应信号 $\overline{\text{INTA}}$，完成一个中断响应周期。

(2) 关中断，即将 IF 标志位置 0，以避免在中断过程中或进入中断服务程序后，再次被其他可屏蔽中断源中断。

(3) 保护处理机的现行状态，即保护现场。包括将断点地址(即下条要取出指令的段基址和偏移量，在 CS 和 IP 内)及标志寄存器 FLAGS 内容压入堆栈。

(4) 在中断响应周期的第二个总线周期中，中断控制器已将发出中断请求外设的中断类型号送到了系统数据总线上，CPU 读取此中断类型号，并根据此中断类型号查找中断矢量表，找到中断服务程序的入口地址，将入口地址中的段基址及偏移量分别装入 CS 及 IP，一旦装入完毕，中断服务程序就开始执行。

可屏蔽外部中断请求中断响应过程如图所示。

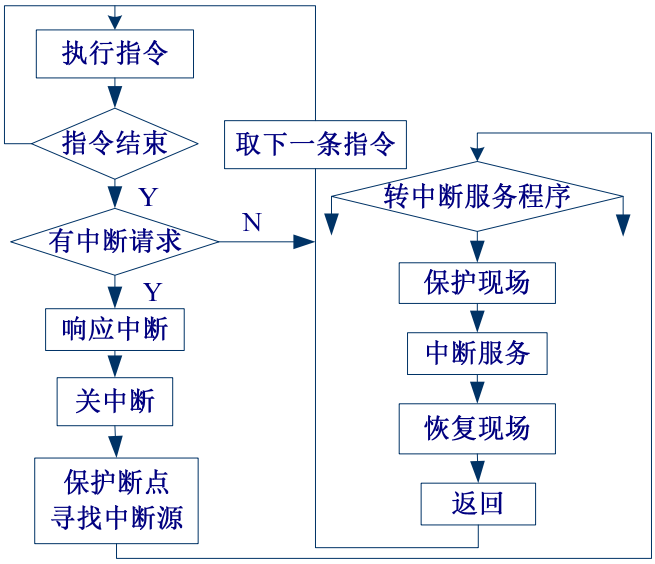


图 5. 47　可屏蔽外部中断请求中断响应过程

不可屏蔽外部中断请求的中断响应

NMI 上中断请求的响应过程要简单一些。只要 NMI 上有中断请求信号(由低向高的正跳变，两个以上时钟周期)，CPU 就会自动产生类型号为 2 的中断，并准备转入相应的中断服务程序。与可屏蔽中断请求的响应过程相比，它省略了第 1 步及第 4 步中的从数据线上读中断类型号，其余步骤相同。

NMI 上中断请求的优先级比 INTR 上中断请求的优先级高，故这两个引脚上同时有中断请求时，CPU 先响应 NMI 上的中断请求。

内部中断的中断响应

内部中断是由 CPU 内部特定事件或程序中使用 INT 指令触发，若由事件触发，则中断类型号是固定的；若由 INT 指令触发，则 INT 指令后的参数即为中断类型号。故中断发生时 CPU 已得到中断类型号，从而准备转入相应中断服务程序中去。除不用检测 NMI 引脚外，其余与不可屏蔽外部中断请求的中断响应相同。

3. 中断处理

中断处理的过程就是 CPU 运行中断服务程序的过程，这一步骤对所有中断源都一样。所谓中断服务程序，就是为实现中断源所期望达到的功能而编写的处理程序。中断服务程序一般由 4 部分组成：保护现场、中断服务、恢复现场、中断返回。所谓保护现场，是因为有些寄存器可能在主程序被打断时存放有用的内容，为了保证返回后不破坏主程序在断点处的状态，应将有关寄存器的内容压入堆栈保存。中断服务部分是整个中断服务程序的核心，其代码完成与外设的数据交换。恢复现场是指中断服务程序完成后，把原先压入堆栈的寄存器内容再弹回到 CPU 相应的寄存器中。有了保护现场和恢复现场的操作，就可保证在返回断点后，正确无误地继续执行原先被打断的程序。中断服务程序的最后部分是一条中断返回指令 IRET。

4. 中断返回

在中断服务程序的最后，应安排一条中断返回指令 IRET。该指令完成如下功能：

(1) 从栈顶弹出一个字→IP。

(2) 再从栈顶弹出一个字→CS。

(3) 再从栈顶弹出一个字→FLAGS。

习　题

1. 8086CPU 内部由哪两部分组成？它们的主要功能是什么？

2. 8086CPU 中有哪些寄存器？各有什么用途？

3. 简要解释下列名词的意义：CPU，存储器，堆栈，IP，SP，BP，段寄存器，状态标志，控制标志，物理地址，逻辑地址，机器语言，汇编语言，指令，内部总线，系统总线。

4. 要完成下述运算或控制，用什么标志位判别？其值是什么？

（1）比较两数是否相等 （2）两数运算后结果是正数还是负数

（3）两数相加后是否溢出 （4）采用偶校验方式，判定是否要补 1

（5）两数相减后比较大小 （6）中断信号能否允许

5. 8086 系统中存储器采用什么结构？用什么信号来选中存储体？

6. 段寄存器装入如下数据，写出每段的起始和结束地址。

7. 根据下列 CS：IP 的组合，求出要执行的下一条指令的存储器地址。

（1）CS：IP=1000H：2000H （2）CS：IP=2000H：1000H

（3）CS：IP=1A00H：B000H （4）CS：IP=3456H：AB09H

8. 求下列寄存器组合所寻址的存储单元地址：

（1）DS=1000H，DI=2000H （2）SS=2300H，BP=3200H

（3）DS=A000H，BX=1000H （4）SS=2900H，SP=3A00H

9. 若当前 SS=3500H，SP=0800H，说明堆栈段在存储器中的物理地址，若此时入栈 10 个字节，SP 内容是什么？若再出栈 6 个字节，SP 为什么值？

10. 某程序数据段中存放了两个字，1EE5H 和 2A8CH，已知 DS=7850H，数据存放的偏移地址为 3121H 及 285AH。试画图说明它们在存储器中的存放情况。若要读取这两个字，需要对存储器进行几次操作？

11. 说明 8086 系统中“最小模式”和“最大模式”两种工作方式的主要区别是什么？

12. 哪个标志位控制 CPU 的 INTR 引脚？

13. 什么叫总线周期？在 CPU 读/写总线周期中，数据在哪个机器状态出现在数据总线上？

14. 8284 时钟发生器共给出哪几个时钟信号？

15. 8086CPU 重新启动后，从何处开始执行指令？

16. 8086CPU 的最小模式系统配置包括哪几部分？

17. 分别说明下列指令的源操作数和目的操作数各采用什么寻址方式。

（1）MOV AX，2408H （2）MOV CL，0FFH （3）MOV BX，[SI]

（4）MOV 5[BX]，BL （5）MOV [BP+100H]，AX （6）MOV [BX+DI]，’$’

（7）MOV DX，ES：[BX+SI] （8）MOV VAL[BP+DI]，DX

（9）IN AL，05H （10）MOV DS，AX

18. 已知：DS=1000H，BX=0200H，SI=02H，内存 10200H～10205H 单元的内容分别为 10H，2AH，3CH，46H，59H，6BH。下列每条指令执行完后 AX 寄存器的内容各是什么？

（1）MOV AX，0200H （2）MOV AX，[200H] （3）MOV AX，BX

（4）MOV AX，3[BX] （5）MOV AX，[BX+SI] （6）MOV AX，2[BX+SI]

19. 设 DS=1000H，ES=2000H，SS=3500H，SI=00A0H，DI=0024H，BX=0100H，BP=0200H，数据段中变量名为 VAL 的偏移地址值为 0030H，试说明下列源操作数字段的寻址方式是什么？物理地址值是多少？

（1）MOV AX，[100H] （2）MOV AX，VAL （3）MOV AX，[BX]

（4）MOV AX，ES：[BX] （5）MOV AX，[SI] （6）MOV AX，[BX+10H]

（7）MOV AX，[BP] （8）MOV AX，VAL[BP][SI]

（9）MOV AX，VAL[BX][DI] （10）MOV AX，[BP][DI]

20. 写出下列指令的机器码

（1）MOV AL，CL （2）MOV DX，CX （3）MOV [BX+100H]，3150H

21. 指出下列指令中哪些是错误的，错在什么地方。

（1）MOV DL，AX （2）MOV 8650H，AX （3）MOV DS，0200H

（4）MOV [BX]，[1200H] （5）MOV IP，0FFH （6）MOV [BX+SI+3]，IP

（7）MOV AX，[BX][BP] （8）MOV AL，ES：[BP] （9）MOV DL，[SI][DI]

（10）MOV AX，OFFSET 0A20H （11）MOV AL，OFFSET TABLE

（12）XCHG AL，50H （13）IN BL，05H （14）OUT AL，0FFEH

22. 已知当前数据段中有一个十进制数字 0～9 的 7 段代码表，其数值依次为 40H，79H，24H，30H，19H，12H，02H，78H，00H，18H。要求用 XLAT 指令将十进制数 57 转换成相应的 7 段代码值，存到 BX 寄存器中，试写出相应的程序段。

23. 已知当前 SS=1050H，SP=0100H，AX=4860H，BX=1287H，试用示意图表示执行下列指令过程中，堆栈中的内容和堆栈指针 SP 是怎样变化的。

    PUSH AX

    PUSH BX

    POP BX

    POP AX

24. 下列指令完成什么功能？

（1）ADD AL，DH （2）ADC BX，CX （3）SUB AX，2710H

（4）DEC BX （5）NEG CX （6）INC BL

（7）MUL BX （8）DIV CL

25. 已知 AX=2508H，BX=0F36H，CX=0004H，DX=1864H，求下列每条指令执行后的结果是什么？标志位 CF 等于什么？

（1）AND AH，CL （2）OR BL，30H （3）NOT AX

（4）XOR CX，0FFF0H （5）TEST DH，0FH （6）CMP CX，00H

（7）SHR DX，CL （8）SAR AL，1 （9）SHL BH，CL

（10）SAL AX，1 （11）RCL BX，1 （12）ROR DX，CL

26. 编程将 AX 寄存器中的内容以相反的顺序传送到 DX 寄存器中，并要求 AX 中的内容不被破坏，然后统计 DX 寄存器中‘1’的个数是多少。

27. 设 CS=1200H，IP=0100H，SS=5000H，SP=0400H，DS=2000H，SI=3000H，BX=0300H，（20300H）=4800H，（20302H）=00FFH，TABLE=0500H，PROG_N 标号的地址为 1200：0278H，PROG_F 标号的地址为 3400H：0ABCH。说明下列每条指令执行完后，程序将分别转移到何处执行？

（1）JMP PROG_N

（2）JMP BX

（3）JMP [BX]

（4）JMP FAR PROG_F

（5）JMP DWORD PTR [BX]

如将上述指令中的操作码 JMP 改成 CALL，则每条指令执行完后，程序转向何处执行？并请画图说明堆栈中的内容和堆栈指针如何变化。

28. 在下列程序段括号中分别填入以下指令

（1）LOOP NEXT （2） LOOPE NEXT （3） LOOPNE NEXT

试说明在这三种情况下，程序段执行完后，AX，BX，CX，DX 的内容分别是什么。

START: MOV AX,01H

MOV BX,02H

MOV DX,03H

MOV CX,04H

NEXT: INC AX

ADD BX,AX

SHR DX,1

(　　)

29. 用 1024×1 位的 RAM 芯片组成 16K×8 位的存储器，需要多少芯片？在地址线中有多少位参与片内寻址？多少位组合成片选择信号？（设地址总线为 16 位）

30. 现有一存储体芯片容量为 512×4 位，若要用它组成 4KB 的存储器，需要多少这样的芯片？每块芯片需要多少寻址线？整个存储系统最少需要多少寻址线？

31. 利用 1024×8 位的 RAM 芯片组成 4K×8 位的存储器系统，试用 A15～A12 地址线用线性选择法产生片选信号，存储器的地址分配有什么问题，并指明各芯片的地址分配。

32. 设计一个 64K×8 存储器系统，采用 74LS138 和 EPROM2764 器件，使其寻址存储器的地址范围为 40000H～4FFFFH。

33. 用 8K×8 位的 EPROM2764、8K×8 位的 RAM6264 和译码器 74LS138 构成一个 16K 字 ROM、16K 字 RAM 的存储器子系统。8086 工作在最小模式，系统带有地址锁存器 8282，数据收发器 8286。画出存储器系统与 CPU 的连接图，写出各块芯片的地址分配。

34. 什么叫中断？什么叫可屏蔽中断和不可屏蔽中断？

35. 列出微处理器上的中断引脚和与中断有关的指令。

36. 8086CPU 可以引入哪些中断？

37. 中断向量表的作用是什么？它放在内存的什么区域内？中断向量表中的什么地址用于类型 3 的中断？

38. 设类型 2 的中断服务程序的起始地址为 0485：0016H，它在中断向量表中如何存放？

39. 若中断向量表中地址为 0040H 中存放 240BH，0042H 单元里存放的是 D169H，试问：（1）这些单元对应的中断类型是什么？（2）该中断服务程序的起始地址是什么？

40. 简要说明 8086 响应类型 0～4 中断的条件是什么？

41. 8086CPU 响应中断的条件是什么？简述中断处理过程。

42. 中断服务子程序中中断指令 STI 放在不同位置会产生什么不同结果？中断嵌套时，STI 指令应如何设置？

43. 假定中断类型号 15 的中断处理程序的首地址为 ROUT15，编写主程序为其建立一个中断向量。

44. 8086CPU 如何获得中断类型号？

45. 给定 SP=0100H，SS=0500H，PSW=0240H，在存储单元中已有内容为 00024H)=0060H，(00026H)=1000H，在段地址为 0800H 及偏移地址为 00A0H 的单元中有一条中断指令 INT 9，试问执行 INT 9 指令后，SP、SS、IP、PSW 的内容是什么？栈顶的三个字是什么？

参考文献

[1] 徐晨，陈继红，王春明等. 微机原理及应用. 高等教育出版社. 2004

[2] 葛纫秋，韩宇龙，武梦龙. 微型计算机结构与编程. 高等教育出版社. 2005

[3] 冯博琴主编. 吴宁，陈文革，张建编著. 微型计算机硬件技术基础. 高等教育出版社. 2003

[4] 周荷琴，吴秀清. 微型计算机原理与接口技术. 中国科学技术大学出版社. 2008

[5] 陈光军，傅越千. 微机原理与接口技术. 北京大学出版社. 2007

第6章. 8086/8088 汇编语言程序设计

汇编语言(Assembly Language)是用助记符、符号地址、标号等编写程序的语言，是机器语言的一种符号表示，主要特点是可以使用助记符来表示机器指令的操作码和操作数，可以用标号和符号来代替地址、常量和变量。助记符一般都是表示一个操作的英文单词缩写，易于记忆和识别。程序员不必具体安排程序在存储器中的物理位置。

计算机程序设计语言一般可分为机器语言、汇编语言和高级语言三种不同层次的语言。用汇编语言编写的程序叫做汇编语言源程序（简称源程序），指令系统中的每条指令都能作为源程序的基本语句。汇编语言是和机器语言密切相关的，是面向机器的语言。CPU 不同的计算机有着不同的汇编语言。汇编语言源程序不能直接在计算机上运行，需要翻译成机器语言程序（目标程序）。将汇编语言源程序翻译成机器语言程序的过程叫汇编，完成汇编任务的程序叫汇编程序，它是一种计算机应用程序。图 6-1-为汇编语言程序的建立与汇编过程。

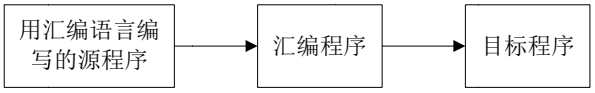


图 6-1　汇编语言程序的建立与汇编过程

汇编程序除完成将汇编语言源程序翻译成机器语言程序外，还有以下任务：按用户要求自动分配存储区；自动把各种进制数转换成二进制数；计算表达式的值；对源程序进行语法检查且给出语法出错信息。基本汇编程序有 ASM（Assembler），进一步允许在源程序中把一个指令序列定义为一条宏指令的汇编程序，叫宏汇编 MASM（Macro Assembler）。8086 系统中常用的汇编程序是 MASM6.X 版本，因此除了指令系统外，还要了解 MASM 中的标号、表达式、伪指令，必须按 MASM 中规定的格式来编写源程序，才能正确汇编成可执行程序。

6.1. 汇编语言源程序的格式

6.1.1. 8086/8088 汇编语言程序的一个例子

我们先介绍一个例子来说明 8086/8088 汇编语言的格式。

【例 6-1】在屏幕上显示并打印字符串“This is a sample program.”

```
DATA    SEGMENT                          ;数据段
      DA1     DB     'This is a sample program.'
              DB     0DH, 0AH, '$'
DATA    ENDS
STACK   SEGMENT
        ST1           DB     100 DUP(?)
STACK   ENDS
```

```
CODE    SEGMENT                         ;代码段
        ASSUME CS: CODE, DS:DATA, SS:STACK
MAIN    PROC    FAR
START:   MOV    AX, STACK               ;送堆栈段段地址
          MOV   SS, AX
PUSH    DS                      ;返回 DOS 用
MOV     AX, 0
          PUSH  AX
          MOV   AX, DATA                ;送数据段段地址
          MOV   DS, AX
          MOV   AH, 9                   ;DOS 9 号功能调用，显示字符串
          MOV   DX, OFFSET DA1
          INT   21H
          RET
MAIN    ENDP
CODE    ENDS
  END   START
```

从上面的例子可以看出，汇编语言程序的结构是分段结构形式，一个源程序由若干逻辑段（SEGMENT）组成，每段以 SEGMENT 语句开始，以 ENDS 结束，整个源程序以 END 结束。一个语句行是由 4 部分组成的，即

标号	操作码	操作数(操作数地址)	；注释

各部分之间至少要用一个空格作为间隔，IBM 宏汇编对于语句行的格式是自由的，但格式化的语句会带来有利的方便。

6.1.2. 8086/8088 汇编语言程序格式

语句类型

8086 汇编语言有两种基本语句：指令性语句和伪指令语句。

（1）指令性语句

指令性语句产生目标代码，即 CPU 能执行的，能完成特定功能的语句。指令性语句由一条 8086 的指令和它的标号及注释三部分组成。标号和注释是任选的。在汇编时，一条指

令性语句被翻译成对应的机器码，对应机器的一种操作。例如：

INIT：　MOV　SI，OFFSET　INBUFF　；设置输入缓冲区指针

INIT 为该语句的标号，指令性语句的标号后面有冒号“ ：”，分号后面是注释内容，说明在程序中该语句的功能。中间的是 8086 的一条数据传送指令。

（2）伪指令语句

伪指令语句不产生目标代码，仅仅在汇编过程中告诉汇编程序应如何汇编。例如，告诉汇编程序已写出的源程序有几个段，定义变量、过程、给变量分配存储单元等。

伪指令语句由一条伪指令和它的标号及注释三部分组成。但伪指令标号后面不能有冒号，这是它和指令性语句的一大差别。例如：

SUM　DB　?　；为符号 SUM 保留一个字节

这是一条完整的伪指令语句，这条语句经过汇编以后，为 SUM 分配一个字节单元，SUM 是标号，它代表由伪指令 DB 分配的那个单元的符号地址，又叫做变量名。在机器代码中，这条语句不会出现，它的功能在汇编时全部完成。

语句的格式

指令性语句和伪指令语句格式类似，一般语句由 1～4 部分构成。格式如下：

［名字］助记符（操作码）［操作数］［；注释］

（3）名字（标号）

名字是一个符号，它表示本条语句的符号地址。一般说来名字可以是标号或变量，统称标识符，它是由字母 A～Z、a～ z 和数字 0～9 及专用字符（？、.、@、_、$）组成。

标识符的命名必须满足如下规则：

①　除数字 0～9 外，其它字符都可放在第一个位置。

②　字符“.”只能出现在名字的第一个位置，其它位置不能出现。

③　名字最长由 31 个字符组成。

指令性语句中的标号实质上是指令的符号地址，如果一条指令前有一个标号，程序中其它地方就可以引用这个标号，指令标号后通常有一个冒号。伪指令语句中的名字可以是变量名、段名、过程名，与指令性语句中的标号不同的是这些伪指令中的名字不是任选的，有的必须有名字，有的必须无名字，标号后面没有冒号。

很多情况下伪指令语句中的名字是变量名，代表存储器中一个数据区的名字。变量的类型有字节、字、双字和四字。

标号和变量都有三种属性：段、偏移及类型。

①　段属性

该属性定义了标号和变量的段起始地址，此值必须在一个段寄存器中。标号的段是它所出现的那个代码段，所以由 CS 表示。变量的段可以是 DS、ES、SS、CS，通常是由 DS 和 ES 指示。

② 偏移属性

该属性表示标号或变量相距段起始地址的字节数，该数是一个 16 位无符号数。

③ 类型属性

该属性对于标号而言，是用于指示该标号是在段内引用还是在其它段中引用。标号的类型有 NEAR（段内引用）和 FAR（段外引用）；对于变量其类型属性说明变量有几个字节的长度。这一属性由定义变量的伪指令 DB、DW 或 DD 确定。DB 把变量定义为字节型，DW 把变量定义为字型；DD 把变量定义为双字型。它们的类型分别为 BYTE（字节）、WORD（字）和 DWORD（双字）。

（4）助记符

指令性语句中的助记符能实现一个具体操作；伪指令语句中助记符是伪指令的定义符，如 DB，SEGMENT，ENDS 等，不能翻译成机器码，只是在汇编中完成相应的控制操作。

（5）操作数

这个字段是助记符的操作对象。操作数一般有常量、寄存器、存储器操作数和表达式等几种形式。

① 常量

8086 汇编可以允许的常量有：数字常量，字符常量。

数字常量可以是二进制常量（以字母 B 结尾，Binary）、十进制常量（以字母 D 结尾或没有任何字母结尾，Decimal）、八进制常量（以字母 O 或 Q 结尾，Octal）和十六进制常量（以字母 H 结尾，Hex），字符常量要由单引号引起来。

【例 6-2】

```
DATA    SEGMENT
    BUF     DB      ‘HOW ARE YOU？’
DATA    ENDS
```

② 寄存器

8086 的寄存器常在操作数域中出现代表某一个操作数，每个寄存器都有一种类型特性，可以确定它是一个字节寄存器还是一个字寄存器。

③ 存储器操作数

存储器操作数包括标号和变量。它可以作为源操作数，也可以作为目的操作数，但不能同时充当源操作数和目标操作数。

标号是可执行的指令性语句的符号地址，它可以作为 JMP 和 CALL 指令的转向目标操作数。

变量通常是指存放在某些存储单元中的数据，这些数据在程序运行期间是可以改变的。变量通过标识符来引用，可以是直接寻址、基址寻址、变址寻址和基址变址寻址方式对其存取。因此，变量可以作为这些存储器访问指令的源或目的操作数。

④ 表达式

表达式由常数、寄存器、标号、变量与一些运算符相组合而成，有数据表达式和地址表达式两种。

（6）注释

分号“ ； ”后面的内容是注释，是语句非执行的部分，仅仅作为编程人员的参考信息，可用英文写也可以用中文写。

6.2. MASM 中的表达式

表达式由运算对象及运算符组成，在汇编时由汇编程序对它进行运算，运算结果作为一个语句中的操作数去使用。运算对象可以是常数、变量或标号，运算结果可以是一个常数，也可以是一个存储器的地址，在此地址中存放了数据(称为变量)或指令(称为标号)。

6.2.1. 算术运算符

算术运算符可以完成算术运算。有+（加法）、-（减法）、*（乘法）、/（除法）、MOD（Module，求余）、SHL(左移)、SHR(右移)共七种运算。这七种运算可直接对数字进行运算。对地址的运算，只用加法和减法才有实际意义，并且要求进行加减的两个地址应在同一段内，否则运算结果便不是一个具有实际意义的有效地址了。

【例 6-3】源程序指令格式如下：

```
BUFFER      DB  2，3，5，7，4
            MOV     AL，BUFFER+3
            MOV     AH，3*2-5 MOD 3
            MOV     BH，00000101B   SHL     4
            MOV     BL，00000101B   SHR     4
```

汇编时，计算表达式形成指令为：

```
            MOV     AL，7
            MOV     AH，1
            MOV     BH，01010000B
            MOV     BL，00000000B
```

6.2.2. 逻辑运算符

逻辑运算符对其操作数进行按位操作。运算后产生一个逻辑运算值，用作指令操作数使用，不影响标志位。对地址不进行逻辑运算，逻辑运算只能用于数字表达式中。

逻辑运算符有：AND（与）、OR（或）、XOR（异或）和 NOT（非）。其中 NOT 是单操作运算符，其它是双操作运算符。

【例 6-4】

```
MOV     AX，0FF00H  AND  10AEH
AND     CX，00FFH   AND  10AEH
```

汇编时，计算表达式形成指令为：

```
MOV     AX，1000H
AND     CX，00AEH
```

从上例可以看出，逻辑运算符与指令系统中的逻辑运算指令是不相同的。逻辑运算是在汇编时完成的，表达式的值由汇编程序确定，而逻辑指令是在程序执行时完成的。

6.2.3. 关系运算符

关系运算符有：EQ（相等 Equal）、NE（不相等 Not Equal）、LT（小于 Little）、GT（大于 Great）、LE（小于或等于）、和 GE（大于或等于）共六种。关系运算符都是双操作运算符，它的运算对象只能是两个性质相同的项目。关系运算的结果只能是两种情况：关系成立或不成立。当关系成立时，运算结果为全 1，否则为全 0。

【例 6-5】

```
MOV     AX，2  LT  5
MOV     AX，2  GT  5
```

汇编时，计算表达式形成指令为：

```
MOV         AX，0FFFFH
MOV         AX，0
```

6.2.4. 分析运算符

分析运算符是对存储器地址进行运算的。它可以将存储器地址的三个重要属性即：段地址、偏移量和类型分离出来，返回到所在位置作操作数使用。分析运算符共有 5 个：SEG、OFFSET、TYPE、SIZE 和 LENGTH。其中 SIZE 和 LENGTH 只对数据存储器操作数有效。

- SEG

  把 SEG 运算符加于一个标号或是变量，求出的是该标号或变量所在的段地址。

  格式：SEG　变量名或标号名

  ```
  MOV     AX，SEG     VAR1          ；把变量 VAR1 所在的段地址送给 AX
  ```

- OFFSET

  把 OFFSET 运算符加于一个标号或是变量，求出的是该标号或变量所在的偏移地址。

格式：OFFSET　变量名或标号名

MOV　AX，OFFSET　VAR1　；把变量 VAR1 所在的偏移地址送给 AX

这条指令等价于：

LEA　AX，VAR1

- TYPE

TYPE 可加在变量或标号的前面，所求出的是这些操作数的类型。

格式：TYPE 变量或标号

表 6-1　TYPE 运算符返回值

	类型	返回值
变	DB	1
	DW	2
量	DD	4
	DQ	8
标	NEAR	-1[FFH]
号	FAR	-2[FEH]

【例 6-6】有如下程序段：

```
A1      DB      20H, 30H
A2      DW      0438H
A3      DD      ?
L1:     MOV     AH, TYPE A1
        MOV     BH, TYPE A2
        ADD     AL, TYPE A3
        MOV     BL, TYPE L1
```

汇编时，计算表达式形成指令为：

```
MOV     AH, 1
MOV     BH, 2
ADD     AL, 4
MOV     BL, 0FFH
```

- LENGTH

LENGTH 只对用 DUP 定义的变量才有意义，它给出分配给变量的单元（字节、字或双字）数，对其它变量则返回 1。

【例 6-7】有如下程序段：

```
M1      DW      100 DUP(?)
M2      DW      1，  2，  3
M3      DB      ‘ABCD’
MOV     CX, LENGTH M1
MOV     BL, LENGTH M2
MOV     AL, LENGTH M3
```

汇编时，计算表达式形成指令为：

```
MOV     CX, 100
MOV     BL, 1
MOV     AL, 1
```

- SIZE

SIZE 运算符加在变量前，返回变量总的字节数，它的值可由以下公式计算：

SIZE　V=（LENGTH　V）×（TYPE　V）

【例 6-8】对例 6-7 定义的 M1，M2，M3

```
    MOV     CX, SIZE M1
    MOV     BL, SIZE M2
    MOV     AL, SIZE M3
```

汇编时，计算表达式形成指令为：

```
MOV     CX, 200
MOV     BL, 2
MOV     AL, 1
```

6.2.5. 修改属性运算符

这种运算符为存储器地址操作数临时指定一个新的属性，而忽略当前的属性，又称为综合运算符。有六个修改属性运算符：PTR，段属性前缀、THIS，SHORT、HIGH 和 LOW。

- PTR

格式：　类型/距离　PTR　变量或标号

功能：将 PTR 左边的类型属性赋给右边的变量或标号。PTR 本身并不分配存储单元，

仅给已分配的存储单元赋予新的属性，这样可以保证运算时操作数类型的匹配，常与类型 BYTE、WORD、NEAR、FAR 等连用。

【例 6-9】

```
DATA1   DB      10H, 20H, 30H                   ;数据定义
DATA2   DW      4023H, 0A845H
MOV     BX，WORD  PTR DATA1     ;2010H 传送到 BX
MOV     AL，BYTE  PTR DATA2     ;23H 传送到 AL
MOV     WORD PTR[BX], 10H       ;[BX],[BX+1]←0010H
```

- THIS

格式：变量/标号　EQU THIS 类型/距离

功能：将 EQU THIS 右边的类型／距离属性，赋给左边的变量／标号，该变量或标号的段地址和偏移地址与下一个存储单元的地址相同

【例 6-10】

```
MY_BYTE   EQU THIS    BYTE
MY_WORD   DW 1122H
          ……
MOV     AL, MY_BYTE       ;将 22H 传送给寄存器 AL
          MOV     BX, MY_WORD       ;将 1122H 传送给寄存器 BX
```

- 段属性前缀

8086 寻址方式中，有一些默认段寄存器的情况。例如，如果用 BP 作为基址寻址的单元，则可表明此单元位于 SS 段；如果用 BX 作为基址寻址的单元，则可表明此单元位于 DS 段。但在某些特例下，需要进行段超越寻址，应使用段属性前缀。

【例 6-11】

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

这条指令是把 ES 段中的偏移地址为 BX 的单元中的字送 AX，而不是到 DS 段去寻址。

- SHORT

格式：JMP　SHORT 标号

功能：指定转移的距离属性为短，实际转移范围为土 127 字节。

- HIGH 和 LOW 运算符

HIGH 和 LOW 被称为字节分离符。它们是将一个 16 位数的高字节和低字节分离出来。

【例 6-12】

```
K1      EQU     0ABCDH
K2      EQU     1234H
        MOV     AH, HIGH K1     ;AH←0ABH
        MOV     BL, LOW K2      ;BL←34H
```

6.3. 伪指令语句

6.3.1. 符号赋值语句

■ 等值语句 EQU

格式：符号　EQU　表达式

功能：用来给变量，标号，常数，指令，表达式等定义一个符号名，在同一个程序模块中不能重新定义。

【例 6-13】

```
A   EQU     7           ;将 7 赋予符号名 A
B       EQU     A-2     ;将 A-2 的值 5 赋予符号名 B
COUT    EQU     CX      ;将 COUT 作为寄存器 CX 的同义名
```

■ 等号 =

此语句与 EQU 类似，其最大特点是能对符号进行再定义，而 EQU 则不能对符号进行再定义。如下例所示。

【例 6-14】

```
COUNT  =100
COUNT  =100+10              ；正确
COUNT  EQU     100
COUNT  EQU     100+10       ；错误
```

■ PURGE

一个符号经过 EQU 赋值后，在整个程序中，这个符号不能再重新赋值，若以后不再需要了，就可以用 PURGE 来解除。

格式：PURGE　符号 1，符号 2，……，符号 n

注意：PURGE 本身不能有名字（标号）。用 PURGE 解除后的符号，就可以重新赋值。

【例 6-15】

```
COUNT  EQU     100
```

```
PURGE   COUNT
COUNT  EQU      200
```

6.3.2. 数据定义语句

格式 1：变量名　助记符　操作数，操作数，..　　;注释

格式 2：变量名　助记符　n　DUP (操作数，操作数，..)　;注释

功能：将操作数存入变量名指定的存储单元中，或者只分配存储空间不存入数据。

变量名—用符号表示，可省略。汇编程序汇编时将此变量的助记忆符后的第一个字节的偏移地址作为它的符号地址。

助记符—主要有：

DB：用来定义字节。

DW：用来定义字。

DD：用来定义双字。

DQ：用来定义四字。

DT：定义十字节。

操作数—可以是常数，字符串，变量，标号，表达式等。

在格式 2 中，用 n DUP( ) 表示时，n 必须为整数，表示括号中操作数的重复次数。

【例 6-16】数据定义如下：

```
DATA      SEGMENT
              DA1     DB 10H              ;变量 DA1 中装入 10H
              DA2     DW 1122H            ;变量 DA2 中装入 22H, 11H
              DA3     DD 0A0H             ;变量 DA3 中装入 A0H, 00H, 00H, 00H
              ST1     DB  'HOW '          ;将字符串'HOW'的 ASCII 码
;装入 ST1 单元开始的存储单元
              ST2     DB 'OK'             ;将字符串'OK'的 ASCII 码
;装入 ST2 单元开始的存储单元
              ST3     DW 'OK'             ;将字符串'OK'的 ASCII 码
;装入 ST3 单元开始的存储单元
              M       DW   2   DUP(?)  ;保留 2 个字的存储单元空间
              ADR1    DW   ST1         ;将 ST1 的偏移地址赋给字变量 ADR1
```

ADR2　DD　ST2　　;将 ST2 的偏移地址和段地址

;赋给双字变量 ADR2

DATA　　ENDS

图 6-2 指出了例 6-16 在存储器中的存放格式。

【注意：】

定义多字节字符串用 DB，DW 只允许包含两个字符。图 6-2 中指出了 DB 和 DW 定义‘OK’时不同的存放格式。

操作数用?定义不确定值变量，以保留存储空间存放运算结果。

用 DW 和 DD 可以将变量或标号逻辑地址存入存储器。当用 DD 来定义时，原变量或标号的偏移地址存入低位字中，原变量或标号的段基址存入高位字中。

变量	MEMORY	EA
DA1	10H	0000
DA2	22H	0001
	11H	0002
DA3	A0H	0003
	00H	0004
	00H	0005
	00H	0006
ST1	48H	0007
	4FH	0008
	57H	0009
ST2	4FH	000A
	4BH	000B
ST3	4BH	000C
	4FH	000D
M	?	000E
	?	000F
	?	0010
	?	0011

ADR1	07H	0012
	00H	0013
ADR2	0AH	0014
	00H	0015
	DATA	0016
		0017

图 6-2 例 6-16 的存储格式

6.3.3. 过程定义语句

过程定义伪指令有 PROC、ENDP、NEAR 和 FAR。在程序设计中，往往将一些重复出现的语句组定义为子程序。子程序又称为过程，可由 CALL 指令来调用。PROC 和 ENDP 总是成对出现的，这两条伪指令中间的内容就是一个过程。

一个过程到底是作为段间调用的过程还是段内调用的过程呢？在源程序中可以用伪指令 NEAR 或 FAR 指明。若为 NEAR 过程，则只能被本代码段调用；若为 FAR 过程，则可由段间程序调用。过程名由编程者自己取，过程名在程序中可以作为标号使用。过程定义的格式为：

```
过程名  PROC   NEAR/FAR
……
RET
过程名  ENDP
```

注意：每一个过程至少得有一个 RET（Return）语句，整个过程执行的最后一条语句必须是 RET，它是过程返回到调用处的关键语句。当一个程序被定义为过程之后，程序中其它地方就可以用 CALL 指令调用这个过程。

格式为：CALL 过程名。

【例 6-17】过程定义及调用格式

```
    NAME1       PROC
                ……
    CALL        NAME2
                ……
                RET
    NAME1       ENDP
NAME2       PROC
```

……

RET

NAME2 ENDP

6.3.4. 段定义语句

8086 的存储器是分段的，所以 8086 必须按段来组织程序和利用存储器。这就需要有段定义伪指令，段定义主要伪指令有：SEGMENT、ENDS、ASSUME 和 ORG。

■ 段定义语句 SEGMENT/ENDS

段定义伪指令 SEGMENT/ENDS 用于段的定义。用它来指定段的名称和范围，并指明段的定位类型、组合类型和类别。段名由编程者自己取，格式为：

段名 SEGMENT [定位类型][组合类型][‘类别’]

……

段名 ENDS

定位类型、组合类型和类别是赋给段名的属性，加上方括号说明这些属性可以省略，省略表示该程序段与其它段没有联系，是独立的；若不能省略，各属性项的书写顺序不能错，并以空格分界。下面详细介绍段的各种属性。

（1）定位类型

用来规定对段起始边界的要求，可以有 4 种选择：

① PAGE：段起始地址的低 8 位必须为 0，即从一页（Page）的起点开始。

② PARA：段起始地址的低 4 位必须为 0，即从某一节（Paragraph）的边界开始。PARA 为缺省类型。

③ WORD：段起始地址的低 1 位必须为 0，即从偶地址开始。

④ BYTE：段起始地址为任意值，即从任何字节开始。

（2）组合类型

表示本段与其它段的关系，是为链接程序提供信息，可以有 6 种选择：

① NONE：表示本段和其它段逻辑上不发生关系，这是缺省的组合类型。

② PUBLIC：该段与其它模块中的同名段连接时，由低地址到高地址连接起来，组成一个逻辑段，连接次序由连接命令指定，连接时满足定位类型要求。

③ STACK：规定被链接的程序中必须至少有一个具有 STACK 属性的段，即堆栈段。如果多于一个，则在初始化时会将第一个 STACK 段的地址送入 SS 寄存器。而段与段之间的链接与 PUBLIC 同样处理。

④ COMMON：链接程序为其它同名同类别的段指定一个相同的段基址，即这些段是相互重叠的。段的长度取决于最长的 COMMON 段的长度。

⑤ AT 表达式：链接程序将表达式计算出来的 16 位地址作为段地址，但它不能用来指定代码段。

⑥ MEMORY：链接程序把本段定位为几个互联段中地址最高的段。若有多个 MEMORY 段，链接程序认为所遇到的第一个为 MEMORY，其余段则具有 COMMON 属性。

（3）类别

这是编者给各段赋予的一种名字信息。类别必须用单引号引起来。通常使用的类别有‘STACK’，‘CODE’，‘DATA’等。

■ 段分配语句 ASSUME

ASSUME 告诉汇编程序，将某一个段寄存器设为某一个逻辑段的段址，ASSUME 只是通知汇编程序有关段寄存器与逻辑段的关系，并没有给段寄存器赋初值。汇编过程中，汇编程序利用 ASSUME 给出的信息，来检查程序中使用的变量和标号，是否可以通过段寄存器来寻址。

格式：ASSUME　　段寄存器名：段名［，段寄存器名：段名…］

其中，段名为程序中 SEGMENT定义的段的段名，段寄存器名为 CS，DS，SS 和 ES 中的一个。值得注意的是，ASSUME 伪指令只是设定了哪个段寄存器指向哪个段，并没有给各段寄存器装入实际的值。所以，在程序的操作部分，要用指令来完成给段寄存器赋初值。但 CS 寄存器是个例外，CS 的值在程序初始化时由汇编程序自动给出，因此不可以在程序中赋值。堆栈段可以不用 ASSUME，此时利用系统设置的堆栈。

■ ORG 伪指令和地址计数器$

ORG 伪指令格式：ORG　　<表达式>

该语句指定在它之后的代码或数据存放的起始地址的偏移量，以表达式的值作为起始地址，连续存放程序或数据，除非遇到一个新的 ORG 语句。

在使用存储器时，先要给出存储单元地址。汇编程序在汇编时给出一个隐含的地址计数器，$是地址计数器的值，也就是当前所使用的存储单元的偏移地址。

6.3.5. 宏指令

在程序设计中，有时可将一段具有特定功能的代码块定义为一个过程，使整个程序清晰，便于调试，这就是前面所讲的过程。过程仅编写一次，汇编后仅有一段代码，主程序可以多次调用它。由于调用和返回的一系列操作，所以执行的速度相对较慢。在汇编语言源程序中，有的程序部分要多次使用，为了不重复编写可使用一条宏指令代替。这条宏指令通过定义，再经过宏汇编产生所需代码，然后将这些代码嵌在调用处。与过程调用不同，它不使用堆栈，仅减少程序的书写，每调用一次，程序代码就会嵌入一次。

■ 宏定义

8086 宏汇编提供了宏定义伪指令 MACRO/ENDM。

宏定义格式：

宏指令名　　MACRO [<形式参数 1>，<形式参数 2>，……]

<重复使用的语句序列>　　；宏体

ENDM

其中，宏指令名由编程人员自己取，调用时就使用宏指令名来调用宏定义。MACRO/ENDM 要成对使用，宏体就是要用宏指令来代替的程序部分，是一组有独立功能的程序段。形式参数的意义和高级语言一样，它实际上是保留一些位置，在使用宏调用时，用实际的参数来替代。

■ 宏调用

经过宏定义后的宏指令就可以在源程序中被调用。为了与过程调用区别，用宏指令调用称为宏调用。

格式：

宏指令名　[<实际参数 1>，<实际参数 2>，……]

当源程序被汇编时，汇编程序将每个宏调用进行宏展开。宏展开就是宏体取代程序中的宏指令名，且实际参数值取代形式参数。在取代时，实际参数和形式参数是一一对应的。8086 宏汇编不要求实际参数和形式参数个数相等，若调用时实际参数多于形式参数，则多余的实际参数被忽略；反之，形式参数多于实际参数，则多余的形式参数变为“空”。例如：

```
宏定义：    SAVERGE   MACRO     R1，R2，R3，R4
                      PUSH      R1
                      PUSH      R2
                      PUSH      R3
                      PUSH      R4
                      ENDM
宏调用：    SAVERGE   MACRO     AX，BX，CX，DX
```

宏调用是在汇编期间展开的。调用的次数越多，占用的存储空间就越大。因此，宏调用简化了源程序，但并不节省内存空间。而过程是在程序运行期间由主程序调用的，它只占有它自身大小的内存空间，无论被调用多少次，也不会增加程序所占有的内存空间，所以使用过程调用可以节省存储空间。但是，过程在执行时，每调用一次都要先保护断点，程序中还要保护现场，返回时要恢复现场，恢复断点。这些操作增加了额外的时间，因而执行时间长，速度相对较慢。用宏定义和宏调用就可以免去这些额外的执行时间，速度相对较快。宏指令还可以使用参数表，可以使编程更加灵活，这是过程所没有的功能。

6.4. DOS 系统功能调用和 BIOS 中断调用

6.4.1. DOS 系统功能调用

DOS 系统功能调用是 DOS 为用户提供的常用子程序，每个子程序对应一个功能号，可在汇编语言程序中直接调用。这些子程序的主要功能包括：

(1) 设备管理（如键盘、显示器、打印机、磁盘等的管理）

(2) 文件管理和目录操作

(3) 其他管理（如内存、时间、日期等管理）

这些子程序给用户编程带来很大方便，用户不必了解有关的设备、电路、接口等方面的问题，只需直接调用即可。 所有的系统功能调用格式都是一致的，按下面步骤进行：

(1) 功能号→AH

(2) 入口参数→指定寄存器

(3) INT 21H

用户只须给出以上三方面信息，DOS 就可根据所给信息自动转入相关子程序执行。

常用的系统功能调用：

1．键盘输入

(1) 1 号调用—从键盘输入单个字符并回显

调用格式：

```
MOV     AH, 1
INT     21H
```

功能： 等待从键盘输入一个字符并送入 AL。执行时系统将扫描键盘，等待有健按下，一旦有健按下，就将其字符的 ASCII 码读入，先检查是否 Ctrl-Break，若是，退出命令执行；否则将 ASCII 码送 AL，同时将该字符送显示器显示。

(2) 10 号调用—从键盘输入字符串

功能： 将从键盘输入的字符串送入内存的输入缓冲区，同时送显示器显示。

系统功能调用前，应先在数据段定义一个缓冲存储区。缓冲存储区的第一个字节存放由用户自己确定的缓冲区长度；第二个字节由系统填入实际存放的字符个数；第三个字节开始的空间存放从键盘输入的字符串。

系统功能调用时，要求 DS：DX 指向输入缓冲区。执行时，逐一读入键盘输入码，存入缓冲区自第三个单元开始的存储区，直至遇到回车符为止。但键入的字符不包括回车符。若实际输入的字符数少于定义的字符数，缓冲区将多余的空间填零；若实际输入的字符数多于定义的字符数，则多余的输入字符被丢掉。

【例 6-18】

```
DATA        SEGMENT
MAXLEN      DB 100              ;缓冲区长度
ACLEN       DB ?                ;由系统填入实际存放的字符个数
STRING      DB 100 DUP(?)       ;存放从键盘输入的字符串
```

```
DATA        ENDS
CODE        SEGMENT
… …
MOV AX, DATA
MOV DS, AX
… …
LEA DX, MAXLEN                ; DS：DX 指向输入缓冲区
MOV AH, 10
INT 21H
… …
CODE        ENDS
```

2．显示输出

(1) 2 号调用—在显示器上显示输出单个字符

调用格式：

```
MOV     DL, 待显示字符的 ASCII 码
MOV     AH, 2
INT     21H
```

功能：将 DL 中的字符送显示器显示。

【例 6-19】显示输出大写字母 A

```
MOV     DL, 'A'
MOV     AH, 2
INT     21H
```

(2) 9 号调用—在显示器上显示输出字符串

调用格式：

```
LEA     DX, 字符串首偏移地址
MOV     AH, 9
INT     21H
```

功能：将当前数据区中 DS：DX 所指向的以'$'结尾的字符串送显示器显示。

【例 6-20】在显示器上显示字符串“YOU ARE SUCESSFUL!”

```
DATA    SEGMENT
STRING      DB   ' YOU ARE SUCESSFUL! $ '
DATA    ENDS
CODE    SEGMENT
… …
MOV AX, DATA
MOV DS, AX
… …
LEA DX, STRING
MOV AH, 9
INT 21H
… …
CODE    ENDS
```

说明：若希望显示字符串后，光标可自动回车换行，可在定义字符串时作如下更改：

STRING DB ' YOU ARE SUCESSFUL! ' ，0AH，0DH，' $ ' ；在字符串结束前加回车换行的 ASCII 码 0AH，0DH

3．打印输出

功能调用号 5，向标准打印输出设备输出一个字符（ASCII 码），入口参数是放入 DL 的将要输出的字符的 ASCII 码。

【例 6-21】

```
MOV     DL，‘a’              ；待打印字符送给 DL
MOV     AH，5
INT     21H
```

【例 6-22】

编程完成如下功能：在屏幕上显示一行提示信息：“WHAT IS YOUR NAME？”接受用户从键盘输入的信息，并存入内存缓冲区。

汇编语言程序如下：

```
DATA    SEGMENT
```

```
BUF     DB  100                 ;缓冲区长度
DB      ?               ;实际输入字符数
DB      100 DUP(?)  ;输入的字符串
M       DB  'WHAT IS YOUR NAME?$'
DATA    ENDS
CODE    SEGMENT
ASSUME      CS:CODE, DS:DATA
START:  MOV AX,DATA
MOV DS, AX
LEA DX, M                       ;输出'WHAT IS YOUR NAME?'
MOV AH, 9
INT 21H
MOV DX, OFFSET BUF          ;从键盘输入
MOV AH, 10
INT 21H
MOV AH, 4CH                 ;返回 DOS
INT 21H
CODE    ENDS
END     START
```

6.4.2. BIOS 中断调用

BIOS 的全称是 ROM-BIOS—ROM Basic I/O System（只读存储器基本输入输出系统）。它是一组固化到微机主板上一个 ROM 芯片上的子程序，主要功能包括：

(1)驱动系统中所配置的常用外设（即驱动程序），如显示器、键盘、打印机、磁盘驱动器、通信接口等。

(2)开机自检，引导装入。

(3)提供时间、内存容量及设备配置情况等参数。

使用 BIOS 中断调用与 DOS 系统功能调用类似，用户也无须了解相关设备的结构与组成细节，直接调用即可。 两者相比较，BIOS 可更直接地控制外设，故能完成更复杂的输入/输出操作；而 DOS 操作对硬件依赖性少，比相应的 BIOS 操作简单，因此在二者能完成同样功能时，应尽量使用 DOS 功能调用。

用户在汇编语言程序中可使用软中断指令"INT n"调用 BIOS 程序，其中 n 是中断类型

码。常用的 BIOS 程序的功能与其中断类型码对应关系如表 6-2 所示。

表 6-2 常用 BIOS 中断类型

中断类型码	BIOS 中断调用功能
10H	显示器 I/O 中断调用（即显示器驱动程序）
13H	磁盘驱动程序
14H	通信驱动程序
16H	键盘驱动程序
17H	打印机驱动程序

当某个 BIOS 程序中具有多种不同功能时，用不同的编号—功能号加以区分，并约定功能号存放在寄存器 AH 中。其调用方法与 DOS 功能调用类似：

(1) 功能号→AH

(2) 入口参数→指定寄存器

(3) 指令“INT n”实现对 BIOS 子程序的调用

下面以键盘 I/O 中断调用为例介绍 BIOS 中断调用的方法。

键盘 I/O 中断调用（INT 16H）有三个功能，功能号为 0～2。

(1) AH=0

功能：从键盘读入字符送 AL。

出口参数：（AL）＝键入字符的 ASCII 码；（AH）＝键入字符的扫描码

【例 6-23】

```
MOV     AH, 0
INT     16H
```

调用结果：将键盘输入字符的 ASCII 码送 AL，扫描码送 AH。

(2) AH=1

功能：从键盘读入字符送 AL，并设置 ZF 标志，若按过任一健，则置 ZF=0，否则 ZF=1。

出口参数：　ZF=0，键盘有输入，（AL）＝键入字符的 ASCII 码；

ZF=1，键盘无输入

(3) AH=2

功能：读取特殊功能键的状态。

出口参数：AL 中为各特殊功能键的状态（如图 6-3）。

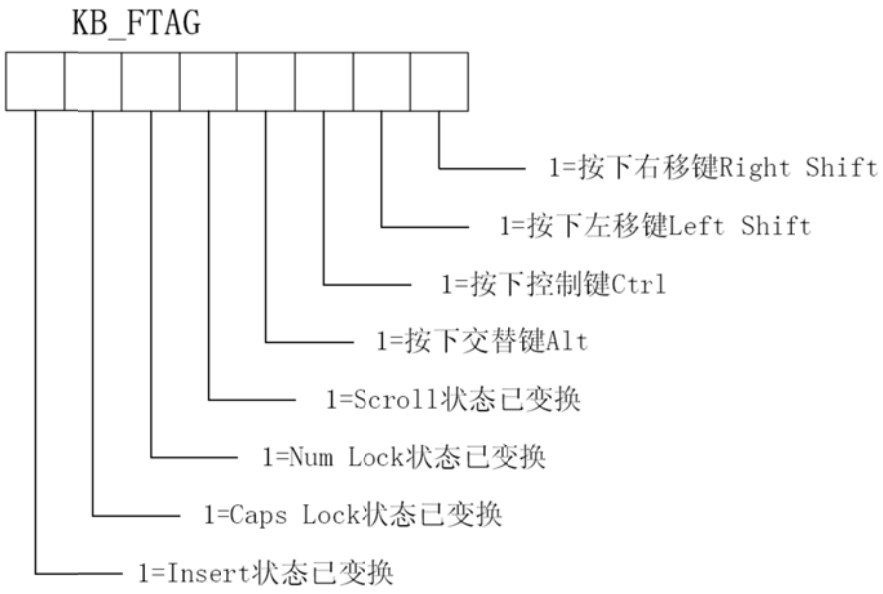


图 6-3 键盘状态字节

6.5. 程序设计方法

设计出一个好的程序不仅要能正常运行，完成要求的功能，还应该具有以下特点：

① 程序结构模块化,简明,易读,易调试,易维护。

② 执行速度快 。

③ 占用存储器空间少。

结构化设计在程序复杂的情况下尤为重要。一般来说设计汇编语言源程序的基本步骤如下：

① 分析问题，抽象出描述问题的数学模型，确定合理算法。

② 绘制程序流程图。

③ 分配存储空间及工作单元（各段位置及段寄存器）。

④ 根据流程图编写程序。

⑤ 静态检查，上机调试。

⑥ 程序运行，结果分析。

在进行汇编语言源程序设计时，通常用到四种程序结构：顺序结构、分支结构、循环结构、子程序结构。四种结构的任意组合和嵌套就构成了结构化的程序。下面分别予以举例说明。

6.5.1. 顺序结构

顺序结构的程序从执行开始到最后一条指令为止，指令指针的内容线性增加，程序一般很简单，没有跳转等语句，例如表达式程序，查表程序就属于这种结构。

【例 6-24】实现两个 32 位无符号数乘法。

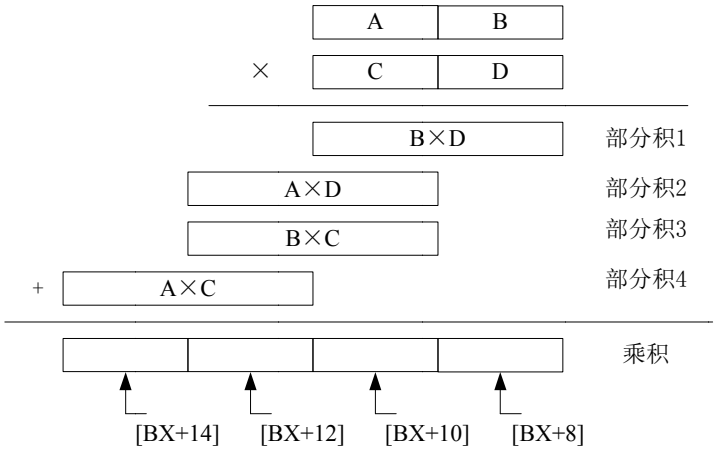


图 6-4　32 位无符号数乘法

程序设计分析：在 8086 中，寄存器是 16 位的，也只有 16 位的运算指令，如果是两个 32 位数相乘就无法直接用指令实现，但可以用 16 位乘法指令做 4 次乘法，然后把部分积相加来实现。

程序清单如下：

```
DATA        SEGMENT
            NUM1        DW 1200H，3400H
            NUM2        DW 5600H，7800H
            MUT         DW 4 DUP（？）
DATA        ENDS
STACK       SEGMENT   PARA    STACK   'STACK'
            DB  100 DUP（？）
STACK       ENDS
CODE        SEGMENT
            ASSUME   CS:CODE，DS:DATA，SS:STACK
BEGIN:      PUSH        DS
            MOV         AX，0
            PUSH        AX
```

```
        MOV     AX，DATA
        MOV     DS，AX
        LEA     BX，NUM1
        MOV     AX，[BX]                ；B→AX
        MOV     SI，[BX+4]              ；D→SI
        MOV     DI，[BX+6]              ；C→DI
        MUL     SI                      ；B*D
        MOV     [BX+8]，AX              ；保存部分积 1
        MOV     [BX+0AH]，DX
        MOV     AX，[BX+2]              ；A→AX
        MUL     SI                      ；A*D
        ADD     [BX+0AH]，AX
        ADC     [BX+0CH]，DX            ；带进位加入积 2 单元中
        MOV     AX，[BX]                ；B→AX
        MUL     DI                      ；B*C
        ADD     [BX+0AH]，AX            ；B*C 加入积单元
        ADC     [BX+0CH]，DX            ；带进位加入积 3 单元中
        ADC     WORD  PTR[BX+0EH]，0    ；带进位加入积 4 单元中
        MOV     AX，[BX+2]              ；A→AX
        MUL     DI                      ；A*C
        ADD     [BX+0CH]，AX
        ADC     [BX+0EH]，DX
        MOV     AH，4CH                 ；返回 DOS
        INT     21H
CODE    ENDS
        END     BEGIN
```

6.5.2. 分支结构

分支结构程序是指程序在按指令先后的顺序执行过程中，遇到不同的计算结果值，需要计算机自动进行判断、选择，以决定转向下一步要执行的程序段。分支程序一般是利用比较、转移指令来实现的。图 6-5 为常见的分支结构。

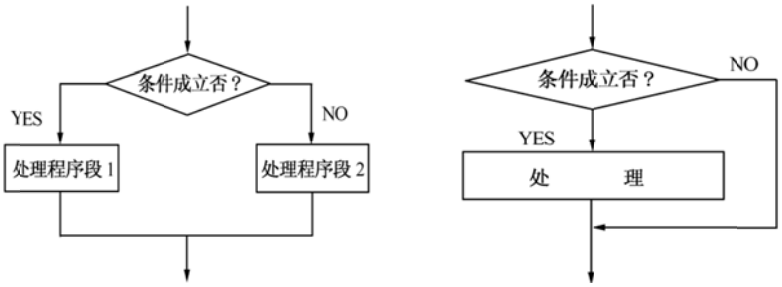


图 6-5 分支结构框图形式

【例 6-25】实现符号函数 $y=\begin{cases}1, x>0\\0, x=0\\-1, x<0\end{cases}$

要实现符号函数，只要把 x 从内存中取出来，执行一次“与”或“或”操作，就可以把 x 的数值特征反映在标志上，再根据标志来转移。流程图如图 6-所示。

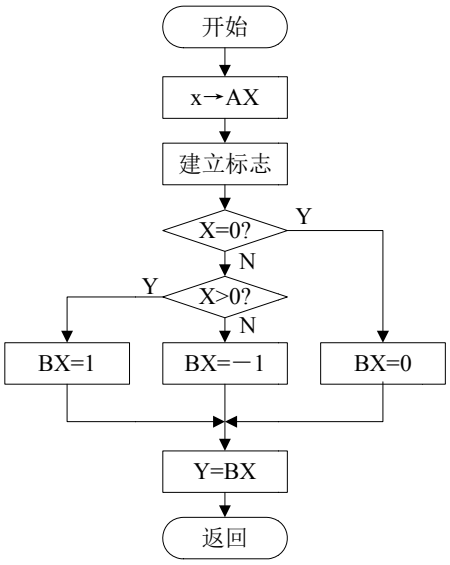


图 6-6 符号函数流程图

源程序如下:

```
DATA        SEGMENT
            X       DW -5
            Y       DW ？
DATA        ENDS
STACK       SEGMENT  PARA  STACK  'STACK'
            DB  100 DUP（？）
```

```
STACK       ENDS
CODE        SEGMENT
            ASSUME    CS:CODE，DS:DATA，SS:STACK
BEGIN:      PUSH    DS
            MOV     AX，0
            PUSH    AX
            MOV     AX，DATA
            MOV     DS，AX
            MOV     AX，X
            AND     AX，AX          ；建立标志
            JZ      ZERO            ；X=0 转 ZERO 执行
            JNS     PLUS            ；X>0 转 PLUS 执行
            MOV     BX，0FFFFH      ；X<0，使 BX =-1
            JMP     DONE
ZERO:       MOV     BX，0
            JMP     DONE
PLUS:       MOV     BX，1
DONE:       MOV     Y，BX           ；存放结果
            MOV     AH，4CH         ；返回 DOS
            INT     21H
CODE        ENDS
            END         BEGIN
```

6.5.3. 循环程序结构

程序中的某些部分要重复执行，设计者不可能也没必要将重复部分反复地书写，那样程序会显得很冗长，这时候就需要用循环结构。循环程序有两种结构形式：一种是“先执行，后判断”结构，另一种是“先判断，后执行”结构。图 6-7 给出了两种循环程序的结构图。

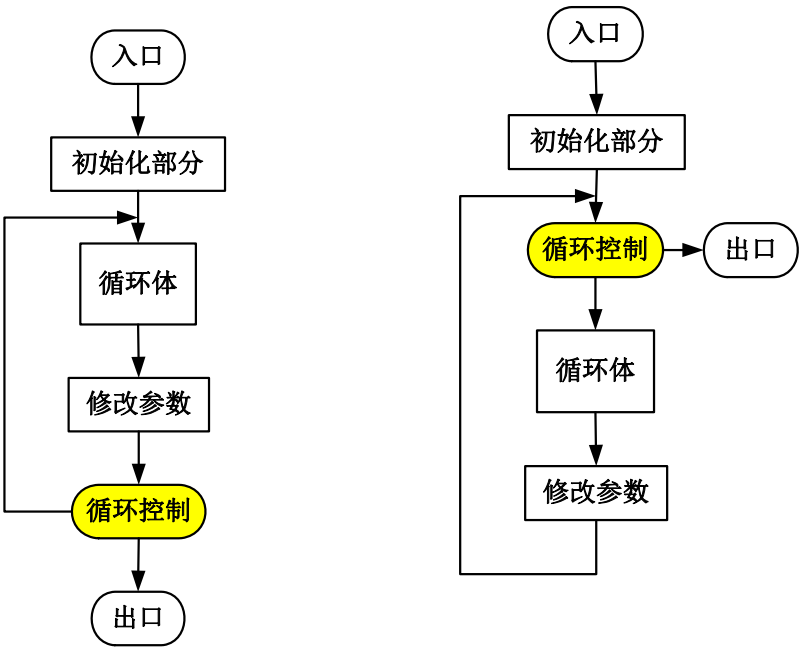


图 6-7 循环程序结构图

无论哪种循环结构，都由以下几部分组成：

⑴初始化:设置循环记数值(次数)和变量初值。

⑵循环体：循环核心，包括循环的全部执行指令。

⑶修改参数：修改源和目的操作数地址。

⑷循环控制：修改记数器值，判断控制条件，决定是否跳转循环。

【例 6-26】 在一串给定个数的数中寻找最大值，放至指定的存储单元。

源程序为：

```
DATA    SEGMENT
        XX      DB  73H，59H，61H，45H，81H
                DB  0FFH，6BH，25H，14H，64H
        YY      DB?
DATA    ENDS
STACK   SEGMENT   PARA    STACK   'STACK'
        DB  100 DUP（？）
STACK   ENDS
```

```
CODE    SEGMENT
        ASSUME    CS:CODE，DS:DATA，SS:STACK
START:  MOV     AX，DATA
        MOV     DS，AX
        MOV     AL，XX
        MOV     BX，OFFSET    XX        ；设地址指针
        MOV     CX，9                   ；循环次数为 9
LOOP1:  INC     BX                      ；指向下一个数
        CMP     AL，[BX]                ；两数比较
        JAE     LOOP2
        XCHG    AL，[BX]                ；把较大的数存于 AL
LOOP2:  DEC     CX                      ；循环次数减 1
        JNZ     LOOP1                   ；循环如果没结束，转至 LOOP1
        MOV     YY，AL                  ；将最大数保存
        MOV     AH，4CH                 ；返回 DOS
        INT     21H
CODE    ENDS
        END     START
```

【例 6-27】将一个带符号字节型数组中的数据按从大到小的顺序排列。

该例使用的是冒泡法排序。

```
DATA        SEGMENT
            BUFF    DB  11H，22H，33H，44H，55H，30H，66H，77H
            COUNT  EQU      $-BUFF
DATA        ENDS
STACK   SEGMENT    PARA        STACK   'STACK'
            DB      100 DUP（？）
STACK   ENDS
CODE        SEGMENT
            ASSUME      CS:CODE，DS:DATA，SS:STACK
```

```
START:  PUSH    DS
        MOV     AX，0
        PUSH    AX
        MOV     AX，DATA
        MOV     DS，AX
        MOV     CX，COUNT-1
LOOP1:  MOV     DX，CX              ；保存循环次数
        MOV     SI，0               ；利用 SI 寻址
LOOP2:  MOV     AL，BUFF[SI]
        CMP     AL，BUFF[SI+1]      ；前数和后数比较
        JGE     DONE                ；前一个数大（或相等）转
        XCHG    AL，BUFF[SI+1]      ；否则交换内存位置
        MOV     BUFF[SI]，AL
DONE:   INC     SI
        LOOP    LOOP2               ；所有数据排列一次
        MOV     CX，DX              ；开始下一次排序
        LOOP    LOOP1
        MOV     AH，4CH             ；返回 DOS
        INT     21H
CODE    ENDS
        END     START
```

6.5.4. 子程序结构

汇编语言中多次使用的程序段可写成一个相对独立的程序段，将它定义为“过程”或称子程序。子程序技术是一种解决重复性问题的重要设计方法，采用子程序结构可以简化源程序书写、提高程序存储效率、减少出错率、增加程序的易读性和可维护性，并且有利于子程序资源的组织和使用。子程序的定义是由过程定义伪指令 PROC 和 ENDP 来完成的。其格式如下：

```
过程名  PROC [NEAR/FAR]
            ……
过程名  ENDP
```

其中 PROC 表示过程定义开始，ENDP 表示过程定义结束。过程名是过程入口地址的符号表示。一般过程名同标号一样，具有三种属性，即段属性、偏移地址属性以及类型属性（NEAR 和 FAR)。如果调用程序和过程在同一代码段中，则使用 NEAR 属性，如果调用程序和过程不在同一代码段中，则使用 FAR 属性。过程调用的指令为 CALL，过程返回的指令为 RET。

设计子程序时，除了必需要考虑程序调用、返回和完成特定功能的指令序列外，还必须注意解决子程序设计中带有共性的一些问题，即:现场保护、参数传递、子程序的嵌套与递归调用、编写子程序说明文档等。

1．现场保护

现场保护的目的是调用子程序之后，能够返回主程序继续执行。因此要对子程序中用到的寄存器，堆栈进行必要的保护。

(1) 寄存器保护。因为汇编语言程序中的主要操作对象是 CPU 中的各寄存器，对那些主程序和子程序中都会用到的一些寄存器要在子程序使用之前进行保护。寄存器保护最好是在子程序中进行，并且在子程序中进行恢复，这样子程序显得更完整。其方法是使用堆栈，由于指令系统中制定了规范的进栈指令 PUSH 和出栈指令 POP，并会自动修改堆栈指针，只要在程序设计中注意 8086/8088 的堆栈是按"后进先出"的原则组织的。

(2) 堆栈保护。子程序是利用调用(CALL)指令和返回(RET)指令来实现正确的调用和返回的。因为 CALL 命令执行时压入堆栈的断点地址就是供子程序返回主程序时的地址，编程时一定要注意子程序的类型属性，即是段内调用还是段间调用。8086/8088 的汇编程序用子程序定义 PROC 的类型属性来确定 CALL 和 RET 指令的属性。如果所定义的子程序是 FAR 属性，那么对它的调用和返回一定都是 FAR 属性；如果所定义的子程序是 NEAR 属性，那么对它的调用和返回也一定是 NEAR 属性。这样用户只是在定义子程序时考虑它的属性，而 CALL 和 RET 指令的属性就可以由汇编程序来确定了。另外，进入子程序后再使用堆栈时也必须保证压入和弹出字节数一致，如果在这里堆栈存取出错，必然会导致返回地址的错误。

2．参数传递

主程序在调用子程序时，经常要向子程序传递一些参数或控制信息，子程序执行后，也常需要把运行的结果返回调用程序，这种信息传递称为参数传递，其常用的方法有寄存器传递、内存固定单元传递、堆栈传递。

(1) 寄存器传递。由主程序将要传递的参数装入事先约定的寄存器中，转入子程序后再取出进行处理。这种方法受 CPU 内部寄存器数量限制，因此只适于传递少量参数的场合，如一些常见的软件延时子程序，均是利用某寄存器传递循环计数器初值。

(2) 通过内存固定单元传递。此方法适用于大量传递参数时使用，它是在内存中开辟特定的一片区域，用于传递参数。主程序和子程序都按事先约定在指定的存储单元中进行数据交换，这种方法要占用一定数量的存储单元。不足之处是信息易被修改,不利于模块化设计。

(3) 通过堆栈实现参数传递。这种方法是先在主程序中把参数和参数地址压入堆栈，在子程序中取出使用，由于堆栈操作不占用寄存器，并且堆栈单元使用后可自动释放，反复使用，便于实现数据隔离和模块化设计。使用这种方法时，当子程序返回后，这些参数就不再有用了，应当丢弃，这时可以利用带立即数的返回指令修改指针，使其指向参数入栈以前的

值。

3．子程序嵌套与递归调用

汇编语言中子程序的嵌套只要堆栈空间允许，一般不受嵌套层次限制。嵌套子程序设计中，应注意寄存器的保护和恢复，避免各层子程序之间寄存器冲突。递归子程序的设计必须保证每次调用都不破坏以前调用时所用的参数和中间结果。为保证每次调用的正确,一般把每次调用的参数、有关寄存器的内容以及中间结果进栈保存。

4．子程序说明文档

一般来说子程序是要反复使用或提供给用户使用，所以编写子程序时应尽量采用较好的算法，使子程序运行速度比较快，又节省内存，同时还应最大限度地满足今后程序维护与使用的需要。在设计子程序的同时就应当建立相应的说明文档，清楚地描述子程序的功能和调用方法。通常子程序说明文档应包括：子程序名称、子程序功能、入口参数、出口参数、工作寄存器、工作单元及最后修改日期等。

【例 6-28】 数据段定义两个数组，编程实现数据段分别求和(不计溢出)。

源程序为：

```
DATA    SEGMENT
        ARY1        DW 100 DUP(?)          ;定义数组 1
        SUM1        DW ?
        ARY2        DW 100 DUP(?)          ;定义数组 2
        SUM2        DW ?
DATA    ENDS
STACK   SEGMENT STACK
        SA          DW 50 DUP(?)
        TOP         EQU LENGTH SA
STACK   ENDS
CODE    SEGMENT
        ASSUME CS:CODE, DS:DATA, SS:STACK
MAIN    PROC    FAR
START:  MOV     AX, DATA
        MOV     DS, AX
MOV     AX, STACK
        MOV     SS, AX
```

```
        MOV     SP, TOP
LEA     SI,ARY1                 ;数组 1 首地址，入口参数
MOV     CX, LENGTH ARY1         ;数组 1 长度，入口参数
    CALL    SUM                     ;调用求和子程序
LEA     SI, ARY2                ;数组 2 首地址，入口参数
MOV     CX, LENGTH ARY2         ;数组 2 长度，入口参数
CALL    SUM                     ;调用求和子程序
MOV     AH,4CH                  ;返回 DOS
INT     21H
RET
MAIN    ENDP
SUM     PROC    NEAR                    ;子程序
        XOR     AX,AX                   ;AX 清 0
L1:     ADD     AX,WORD PTR[SI]         ;加数组元素
        INC     SI
        INC     SI
        LOOP    L1
        MOV     WORD PTR[SI],AX         ;数组和送 SUM
        RET
SUM     ENDP
CODE    ENDS
        END     START
```

本例是通过内存固定单元传递参数的，需要传递的数组保存在内存固定单元，调用前只需将数组偏移地址放入 SI 寄存器，在过程中通过寄存器间接寻址就可以取得内存单元中的操作数，运算结果直接由过程写回到内存单元中，回送给调用程序。

【例 6-29】 通过堆栈传递参数，实现十进制数数组求和，要求主程序和过程不在同一个代码段中，要进行段间调用。

源程序如下：

```
MDATA       SEGMENT
```

```
                ARY1    DB 20 DUP(?)            ;定义数组 1
                SUM1    DW ?
                ARY2    DB 100 DUP(?)           ;定义数组 2
                SUM2    DW ?
MDATA       ENDS
MSTACK      SEGMENT     STACK 'STACK'
                SB      DW 100 DUP(?)
TOP     LABEL WORD
MSTACK      ENDS
MCODE       SEGMENT                             ;主程序
ASSUME CS:MCODE, DS:MDATA, SS:MSTACK
MAIN        PROC    FAR
START:      MOV     AX, MSTACK
MOV     SS, AX
MOV     SP, OFFSET TOP
PUSH    DS
MOV     AX, 0
PUSH    AX
MOV     AX, MDATA
MOV     DS, AX
                MOV     AX,OFFSET ARY1          ; PADD 过程入口参数进栈
                PUSH    AX
                MOV     AX,SIZE ARY1
                PUSH    AX
                CALL    FAR PTR PADD
MOV     AX, OFFSET ARY2
PUSH    AX
MOV     AX, SIZE ARY2
PUSH    AX
CALL    FAR PTR PADD
```

```
RET
MAIN        ENDP
MCODE       ENDS
PCODE       SEGMENT                          ;子过程
ASSUME CS:PCODE, DS:MDATA, SS:MSTACK
PADD        PROC    FAR
                    PUSH    BX               ;寄存器保护
                    PUSH    CX
                    PUSH    BP
                    MOV     BP, SP
                    PUSHF
                    MOV     CX, [BP+10]      ;取数组长度
                    MOV     BX, [BP+12]      ;取数组起始地址
MOV     AX, 0
NEXT:       ADD     AL, [BX]                 ;数组相加
DAA
MOV     DL, AL
MOV     AL, 0
ADC     AL, AH
DAA
MOV     AH, AL
MOV     AL, DL
INC     BX
LOOP    NEXT
                    MOV     [BX], AX         ;数组和送 SUM
                    POPF
                    POP     BP
                    POP     CX
                    POP     BX
                    RET     4                ;返回作废参数
```

```
PADD        ENDP
PCODE       ENDS
END     START
```

说明：

LABEL 伪指令给已经定义的变量或标号取另一个名字，使同一变量或标号在不同地方被引用时可采用不同的名字，具有不同的类型属性。格式为：“名称 LABEL 类型属性”，其功能是为 LABEL 语句下一行所使用的语句中的变量或标号取别名，此别名与原变量或标号具有相同的段地址及偏移地址。

使用堆栈传递参数，调用前要给过程传递两个参数，主程序中将数组的偏移地址值及数组长度压入堆栈，然后调用过程。

段间过程调用，进入过程时要重新分配 CS 段，使之指向当前有效代码段 PCODE；在过程运行中可直接调用堆栈中参数，完成累加运算，并将结果送回指定的存储单元。

过程返回时用返回指令 RET 4，要将堆栈中由 CALL 指令之前传递来的 4 个字节作废。

6.5.5. 程序设计实例

【例 6-30】将内存 BUFFER 中的 16 位二进制数每一位转换为相应的 ASCII 码，存入 STRING 开始的内存中，并显示结果。

源程序如下:

```
DATA        SEGMENT
            BUFFER      DW 1234H
            STR         DB  16 DUP（？）
DATA        ENDS
STACK       SEGMENT   PARA    STACK   'STACK'
            DB  100 DUP（？）
STACK       ENDS
CODE        SEGMENT
            ASSUME      CS:CODE，DS:DATA，SS:STACK
START:      PUSH    DS
            MOV     AX，0
            PUSH    AX
            MOV     AX，DATA
```

```
        MOV     DS，AX
        LEA     DI，STR
        MOV     CX，LENGTH  STR
        PUSH    DI
        PUSH    CX
        MOV     AL，30H
        REP     STOSB                   ；使缓冲区全为 0
        POP     CX
        POP     DI
        MOV     AL，31H                 ；把 1 送入 AL
        MOV     BX，BUFFER
AGAIN:  RCL     BX，1                   ；左移 BX，把相应位送入 CF
        JNC     NEXT                    ；若该位为 0，则转移
        MOV     [DI]，AL                ；若该位为 1，则把 1 送入
NEXT:   INC     DI
        LOOP    AGAIN                   ；没有结束，继续循环
        MOV     AL，'$'
        MOV     [DI]，AL
        MOV     DX，OFFSET      STR     ；显示字符串
        MOV     AH，9
        INT     21H
        MOV     AH，4CH                 ；返回 DOS
        INT     21H
CODE    ENDS
        END     START
```

【例 6-31】实现十六进制数到十进制数的转换。

任意进制到十进制转换可以按权位展开，但用这种方法编程会遇到一些困难。我们从所学过的指令 AAM 中知道，AAM 指令是对两个非压缩的 BCD 码乘法后产生的十六进制结果进行调整，调整方法就是将乘法结果除 10，其商为十进制数的十位，余数为个位；同理，若除 100，其商就是十进制的百位，余数再除 10，其商为十进制数的十位，余数则为个位。

源程序如下：

```
STACK    SEGMENT   PARA   STACK   'STACK'
         DB  100  DUP（？）
STACK    ENDS
CODE     SEGMENT
         ASSUME    CS:CODE，SS:STACK
START:   MOV       AX，STACK
         MOV       SS，AX
         MOV       AX，3FE0H
         MOV       CX，0                ；统计除法次数
         MOV       BX，10
LOP:     MOV       DX，0                ；被除数扩展为 32 位
         DIV       BX
         PUSH      DX                   ；将转换好的数存入堆栈
         INC       CX
         OR        AX，AX               ；转换直到商为 0
         JNZ       LOP
         MOV       AH，4CH
         INT       21H
CODE     ENDS
         END       START
```

【例 6-2】若有一个 ASCII 字符串，从串中取出每一个字符，检查字符中包含 1 的个数，若为偶数，则它的最高位置“0”；否则，最高位置“1”。

源程序如下：

```
DATA     SEGMENT
         STR       DB  'ABCD$'
         COUNT     EQU     $-STR
DATA     ENDS
STACK    SEGMENT   PARA   STACK   'STACK'
```

```
                DB  100  DUP（？）
STACK           ENDS
CODE            SEGMENT
                ASSUME      CS:CODE，DS:DATA，SS:STACK
START:          MOV     AX，DATA
                MOV     DS，AX
                LEA     SI，STR
                MOV     CX，COUNT
LOOP:           LODSB                       ；将字符送至 AL
                AND     AL，AL              ；取标志
                JPE     NEXT                ；若为偶数，则转移
                OR      AL，80H             ；若为奇数，则最高位置 1
                MOV     [SI-1]，AL
NEXT:           DEC     CX
                JNZ     LOOP
                MOV     AH，4CH             ；返回 DOS
                INT     21H
CODE            ENDS
                END     START
```

习题

1.下列语句在存储器中分别为变量分配多少字节空间？并画出存储空间的分配图。

```
VAR1        DB      10，2
VAR2        DW      5DUP（？），0
VAR3        DB      ‘HOW  ARE  YOU？’，‘$’
VAR4        DD      -1，1，0
```

2. 已知内存 DATA 单元存放一个字数据，统计其含有 0 的个数，并送入 RESULT 单元。

3．试编程序将内存从 40000H 到 4BFFFH 的每个单元中均写入 55H，并再逐个单元读出比较，看写入的与读出的是否一致。若全对，则将 AL 置 7EH；只要有错，则将 AL 置 81H。

3. 已知从数据段 DATA 单元开始存放字节型的带符号数 X 和 Y，请设计计算 Y=6X+8 的程序。

4. 已知从数据段 BUF 单元开始存放 15 个字节型数据，请设计程序将其中负数和零分别送往 MINUS 和 ZERO 开始的存储器单元。

5．在当前数据段 4000H 开始的 128 个单元中存放一组数据，试编程序将它们顺序搬移到 A000H 开始的顺序 128 个单元中，并将两个数据块逐个单元进行此较；若有错将 BL 置 00H; 全对则将 BL 置 FFH，试编程序。

6．试编程序，统计由 40000H 开始的 16K 个单元中所存放的字符“A“的个数，并将结果存放在 DX 中。

第7章. 典型接口芯片原理和应用

接口是 CPU 与外部设备之间进行信息交换的必经通道。为了实现接口的功能，要求接口电路应能完成信息缓冲、信息转换、电平转换、数据存取和传送、联络控制等工作，这些工作分别由接口电路的两大部分即与计算机连接的总线接口和与外部设备连接的外设接口来实现。总线接口一般包括内部寄存器、存取逻辑和传送控制逻辑电路等，主要负责数据缓冲、传输管理等工作；而外设接口则负责与外部设备通信时的联络和控制以及电子和信息转换等。本章主要讨论外设接口电路，以下如无特殊说明，所讨论的接口电路均指外设接口。

接口电路从总的功能上可以分为输入接口和输出接口，分别完成信息的输入和输出。从传送方式上，又可分为并行接口和串行接口。另外，从所传送信息的类型上，还可分为数字量的输入/输出(I/O)接口及模拟量的输入 / 输出接口。

近年来，随着超大规模集成电路技术的发展，I/O 接口电路的集成度和智能化程度越来越高，单个接口集成电路芯片的集成度通常超过几万个元器件/片。集成度的提高使得在接口芯片内直接集成 I/O 专用微处理器成为可能，这就是所谓的智能化的 I/O 接口。智能 I/O 接口能够根据主机发送的命令执行相应的控制程序，完成非常复杂的 I/O 操作，这样就大大减轻了 CPU 的负担。I/O 接口的大规模集成化也降低了外围电路的复杂性，从而提高了电路的可靠性。

本章以某些典型的接口芯片为例来介绍接口电路的构成及其具体应用方法。

7.1. 简单 I/O 接口电路及其应用

7.1.1. 接口电路的构成

把信息从外部设备送入 CPU 的接口称为输入接口，而把信息输出到外部设备的接口则称为输出接口。

在需要从外设输入数据时，通常外设的速度相对于 CPU 要慢得多，这意味着数据在外部总线上保持的时间相对较长，所以要求输入接口必须要具有对数据的控制能力，即要在外部数据准备好，CPU 可以读时才允许将数据送上系统数据总线。大多数外设都具有数据保持能力(即 CPU 没有读取时，外设能够保持数据不变)，通常可以仅用三态门缓冲器(简称三态门)作为输入接口。当三态门的控制端信号有效时，三态门导通，该外设就与数据总线连通，CPU 将外设准备好的数据读入；当控制端信号无效时，三态门断开，该外设就从数据总线脱离，数据总线又可用于其他信息的传送。对没有数据保持能力的外设，可在外设与接口之间增加一个锁存器，用外设提供的数据准备好信号把数据保存到锁存器中。

在数据输出时，同样应考虑外设与 CPU 速度的配合问题。要使数据能正确写入外设，CPU 输出的数据一定要能够保持一段时间。一般 CPU 送到总线上的数据只能保持几个微秒甚至更短的时间。相对于慢速的外设，数据在总线上几乎是一闪而逝。因此，要求输出接口必须要具有数据的锁存能力，这通常是由锁存器来实现的。CPU 输出的数据通过总线锁存到锁存器中，并一直保持到被外设取走。

三态门缓冲器和锁存器的控制端一般与 I/O 地址译码输出信号线相连，当 CPU 执行 I/O 指令时，指令中指定的 I/O 地址经译码后即可使控制信号有效，打开三态门(对外设读时)或

将数据锁入锁存器(对外设写时)。

7.1.2. 简单输入接口电路

三态缓冲器常用来构造输入接口。如图 7.1 所示为三态缓冲器芯片 74LS244。该芯片由 8 个三态门构成，它包含两个控制端：$1\overline{G}$ 和 $2\overline{G}$，每个控制端各控制 4 个三态门。当某一控制端有效(低电平)时，相应的 4 个三态门导通；否则，相应的三态门呈现高阻状态(断开)。在实际使用中，可将两个控制端并联，这样就可用一个控制信号来使 8 个三态门同时导通或同时断开。

由于三态门具有控制信号通过的能力，故可利用其作输入接口。利用三态门作为输入接口时，由于三态门没有锁存功能，因此要求外设数据信号的状态能够保持到 CPU 完全读入为止。图 7.2 所示是一个利用一片 74LS244 作为开关量输入接口的例子。在图 7.2 中，74LS244 的输入端接有 8 个开关 S_1～S_8，其输出端接到数据总线上。当 CPU 读该接口(对 80×86 系列 CPU 来说，就是执行 IN 指令)时，总线上的 16 位地址信号通过译码使 $1\overline{G}$ 和 $2\overline{G}$ 有效(逻辑 0)，于是三态门导通，8 个开关的状态经数据线 D_0～D_7 被读入到 CPU 中。这样，就可测量出这些开关当前的状态是打开还是闭合。当 CPU 不访问此接口地址时，$1\overline{G}$ 和 $2\overline{G}$ 2 为高电平(逻辑 1)，则三态门的输出为高阻状态，使其与数据总线断开。

功能表

输入		输出
$\overline{G}$	A	Y
0	0	0
0	1	1
1	×	高阻

图 7.1　74LS244 芯片引出线图

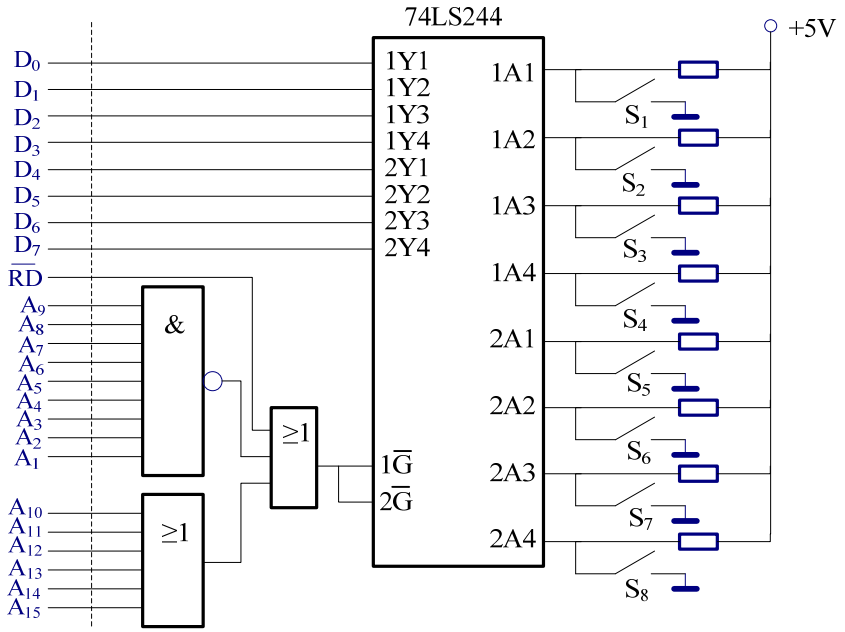


图 7.2　用三态缓冲器 74LS244 构成的简单输入接口

用一片 74LS244 芯片作为输入接口，最多可以连接 8 个开关或其他具有信号保持能力的外设。当然也可只连接少于 8 个的外设而让其他端悬空，对空着未用的引出线，其对应位的数据是任意值，可在程序中可通过逻辑与指令将其屏蔽掉。如果有更多的开关状态(或其他外设)需要输入时，可用类似的方法用两片或更多的芯片组合使用。

例 7.1　输入接口电路如图 7.2 所示，要求编写程序段判断图中的开关的状态。若开关 S_1 闭合，则程序转向标号为 SONE 的程序段执行；若开关 S_2 闭合，则程序转向标号为 STWO 的程序段执行；…；若开关 S_8 闭合，则程序转向标号为 SEIGHT 的程序段执行；对于其他开关状态，程序转向标号为 OTHER 的程序段执行。

解　图 7.2 中三态缓冲器 74LS244 的 I/O 地址采用了部分地址译码——地址线 A_0 没有

参加译码，因此选通 74LS244 的地址为占用了 03FEH 和 03FFH 个地址。其中任何一个地址都可以选中 74LS244 缓冲器。另外，由图 7.2 可以看出，当某个开关闭合时，其对应的输入为低电平(逻辑 0)，当某个开关打开时，其对应的输入为高电子(逻辑 1)。读开关状态的程序段如下：

```
MOV DX, 03FEH

IN   AL, DX

CMP AL, 01H

JZ SONE

CMP AL, 02H

JZ STWO

CMP AL, 04H

JZ STHREE
```

```
CMP AL, 08H
JZ SFOUR
CMP AL, 10H
JZ SFIVE
CMP AL, 20H
JZ SSIX
CMP AL, 40H
JZ SSEVEN
CMP AL, 80H
JZ SEIGHT
JMP OTHER
```

7.1.3. 简单输出接口电路

三态门不具备保存(锁存)数据的能力，它不能直接用于数据输出接口电路。数据输出接口通常是用具有信息存储能力的双稳态触发器来实现。最简单的输出接口可用 D 触发器构成。图 7.3 所示为常用的锁存器 74LS273 引脚和真值表。74LS273 内部包含了 8 个 D 触发器，共有 8 个数据输入端(D_0～D_7)和 8 个数据输出端(Q_0～Q_7)。$\overline{MR}$ 为复位端，低电平有效。CP 为脉冲输入端，在每个脉冲的上升沿将输入端 D_n 的状态锁存在输出端 Q_n，并将此状态保持到下一个时钟脉冲的上升沿。

74LS273 常用来作为简单并行输出接口。另外，使用其中的某一个 D 触发器也可通过软件编程实现简单的串行输出。

真值表

输入			输出
$\overline{MR}$	D_n	CP	Q_n
0	×	×	0
1	↑	1	1
1	↑	0	0

图 7.3　74LS273 引出线排列和真值表

图 7.4 所示的是应用 74LS273 作为输出接口的例子。8 个数据输出端 Q_n 与 8 个发光二极管相连接，8 个数据输入端 D_n 接到数据总线上，地址译码电路的输出接 CP 端，74LS273 的 $\overline{MR}$ 引脚接高电平。

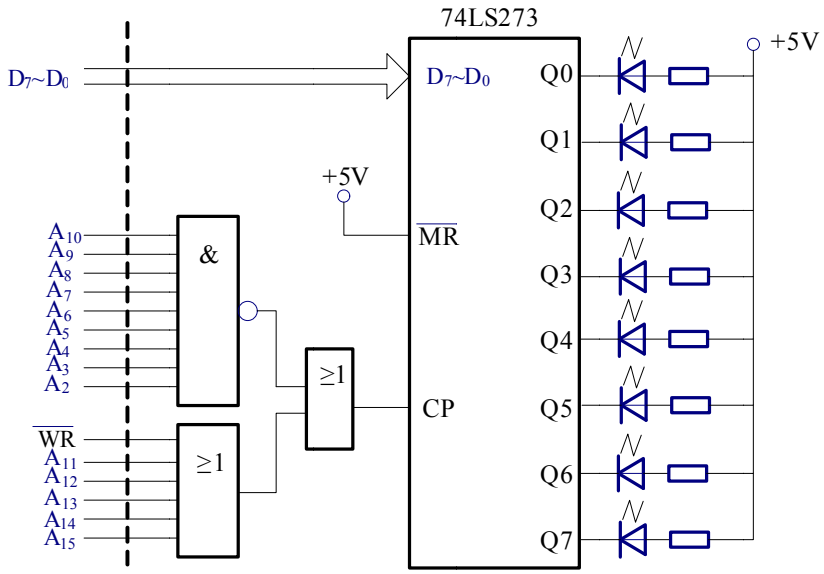


图 7.4　74LS273 用做 LED 显示器的简单输出接口

例 7.2　输出接口电路如图 7.4 所示。要求编写程序段使接到 Q0 端、Q3 端和 Q6 端的发光二极管发光。

解　要使接到 Q0 端、Q3 端和 Q6 端的发光二极管发光，其对应的 Q0 端、Q3 端和 Q6 端应为“0”状态，而其他 Q 端则为“1”状态。由于地址线 A_1、A_0 没有参与地址译码，因此该输出接口的地址为 7FFCH～7FFFH。使 Q0 端、Q3 端和 Q6 端的发光二极管发光的程序段如下：

```
MOV DX, 7FFCH

MOV AL, 01001001B

OUT DX, AL
```

7.1.4. 简单双向接口电路

74LS273 的数据输出端不是三态输出的。只要 74LS273 正常工作，其 Q 端总有一个确定的逻辑状态(0 或 1)输出。因此，74LS273 无法直接用做输入接口，即它的 Q 端不允许直接与系统的数据总线相连接。既可做输入接口又可做输出接口要求接口电路带有三态输出的锁存器。图 7.5 所示为带有三态输出的锁存器 74LS373 及其真值表。它是经常用到的一种电路芯片，从引出线上可以看出，它比 74LS273 多了一个输出允许端 $\overline{OE}$ 。只有当 $\overline{OE}$ =0 时，741S373 的输出三态门才导通；当 $\overline{OE}$ =1 时，则输出三态门呈高阻状态。

真值表

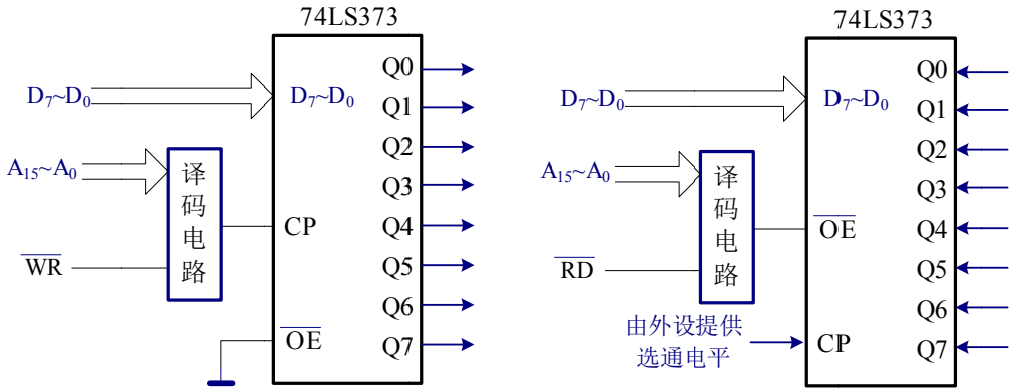
输入			输出
$\overline{OE}$	CP	D_n	Q_n
0	1	1	1
0	1	0	0
0	0	×	Q_{n-1}
1	×	×	高阻

图 7.5　74LS373 引出线排列和真值表

74LS373 在用做输入接口时，CPU 提供的端口地址信号经译码电路接到$\overline{OE}$端，数据由外设提供的选通脉冲(接到 74L5373 的 CP 端)锁存在 74LS373 内部。当 CPU 读该接口时，译码器输出低电平，使 74LS373 的输出三态门打开，数据送到总线上由 CPU 读入。如果把 74LS373 用做输出接口，可将$\overline{OE}$端接地，使其输出三态门一直处于导通状态，这样就与 74LS273 一样使用了。

另外还有一种常用的带有三态门的锁存器芯片 74LS374，它与 74LS373 在结构和功能上完全一样，区别是数据锁存的时机不同，74LS374 是在 CP 脉冲的上升沿将数据锁存。

用 74LS373 分别作为输入接口和输出接口的电路如图 7.6 所示。



(a)　74LS373 用做输出接口　　(b)　74LS373 用做输出接口

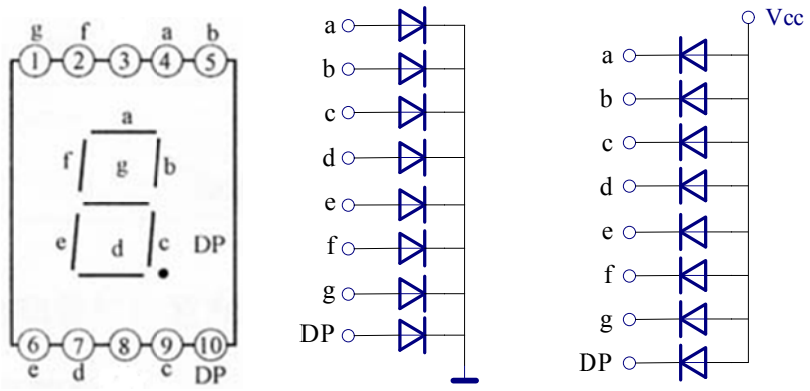
图 7.6　74LS374 用做输入和输出接口

7.1.5. 应用举例

下面举例说明如何利用 74LS244 和 74LS273 作为输入和输出接口，通过编写程序，控制 LED 数码管显示不同的数字或符号。

LED 数码管是将多个码段（二极管）按一定的现状集成在一起，通过控制不同码段发光以显示不同字符的器件，分为共阳极和共阴极两种结构。在封装上有将 1 位、2 位或更多位封装在一起的，如图 7.7 所示。当某一段的发光二极管流过一定电流(例如 10mA 左右)时，

它所对应的段就发光，而无电流流过时，则不发光。不同发光段的组合就可显示出不同的数字和符号。表 7.1 列出了符号“0”～“9”的段码值。



(a)　数码管外形　　(b)　共阴极型　(c)　共阳极型

图 7.7　LED 数码管示意图

表 7.1　常用符号的七段码表

符号	形状	共阴极段码 DPgfedcba	共阳极段码 DPgfedcba
0	0	00111111	11000000
1	1	00000110	11111001
2	2	01011011	10100100
3	3	01001111	10110000
4	4	01100110	10011001
5	5	01101101	10010010
6	6	01111101	10000010
7	7	00000111	11111000
8	8	01111111	10000000
9	9	01101111	10010000

七段数码管作为一种外设与系统总线有多种接口方式，图 7.8 所示电路采用 741S273 作为输出接口，其输出端用集电极开路门 74LS16 来驱动共阳极 LED 数码管。输入接口采用 74LS244。

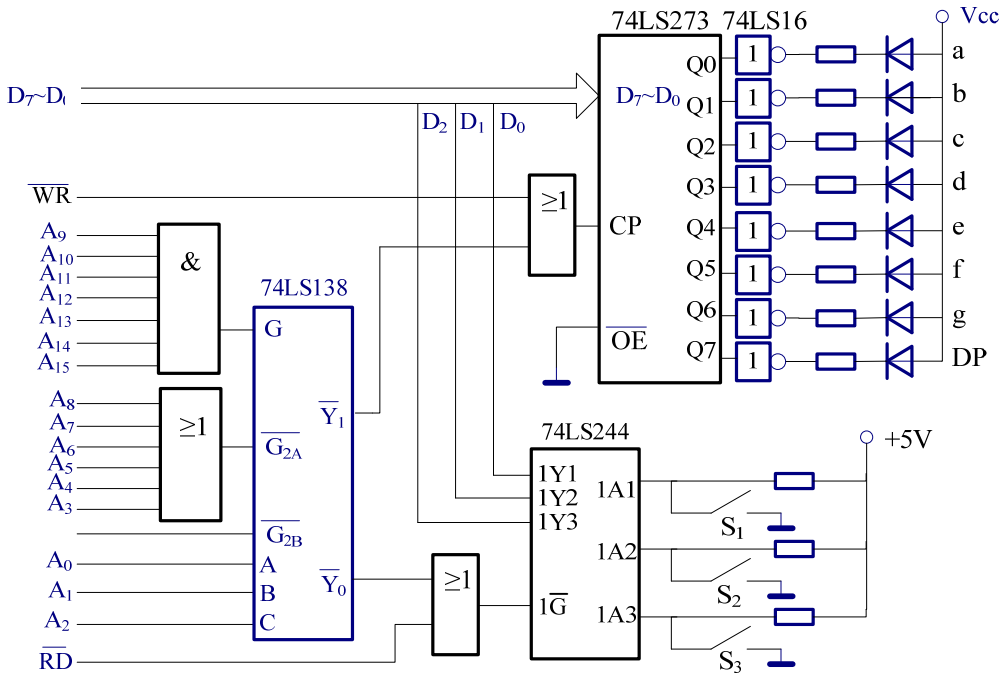


图 7.8　简单接口电路的应用

例 7.3　接口电路如图 7.8 所示，试编程实现如下功能：当开关状态 $S_3S_2S_1$ 为 000（开关全部断开）～111（开关全部合上）时，数码管显示字符“0”～“7”。

解　当开关状态 $S_3S_2S_1$ 为 000 时，CPU 选通 74LS244 后在数据线上的数据为 $D_2D_1D_0$=111。选通 74LS244 的地址为 0FE00H。如果要在数码管上显示字符“0”，则要求选通 74LS273 后数据线上的数据为 D_7～D_0=00111111。选通 74LS273 的地址为 0FE01H。注意：电路采用共阳极 LED 数码管，但在数码管的段码端接有反相驱动器 74LS16，因此段码 DPgfedcba=11000000。开关状态与数码管显示数据见表 7.2。

表 7.2　开关状态与输入输出数据

开关状态 $S_3S_2S_1$	输入数据 $D_2D_1D_0$	显示符号	段码 DPgfedcba
开开开	111	0	00111111B=3FH
开开合	110	1	00000110B=06H
开合开	101	2	01011011B=5BH
开合合	100	3	01001111B=4FH
合开开	011	4	01100110B=66H
合开合	010	5	01101101B=6DH
合合开	001	6	01111101B=7DH
合合合	000	7	00000111B=07H

与硬件电路相配合完成所要求功能的程序段如下：

```
        MOV BX,OFFSET ASCII      ;取字符显示表首地址
EVER: MOV DX, 0FE00H             ;输入端口地址为 0FE00H
        IN AL, DX                ;读入开关状态
        AND AL, 07H              ;取 D2D1D0 位，屏蔽其它位
        XLAT                     ;查表转换
        MOV DX, 0FE01H           ;输出端口地址为 0FE01H
        OUT DX, AL               ;显示字符
        JMP  EVER                ;循环显示
DISP  DB  07H, 7DH, 6DH, 66H, 4FH, 5BH, 06H, 3FH    ;字符显示表
```

从以上的介绍可以看出，三态缓冲器和锁存器在构造上比较简单，使用也很方便，常作为一些功能简单的外部设备的接口电路。但由于它们的功能有限，对较复杂的功能要求就难以胜任，此时可采用可编程数字接口芯片。

7.2. 可编程计数器/定时器 8253 及其应用

7.2.1. 可编程接口芯片概述

简单的接口电路一般只适合于慢速且功能比较简单的外设，难以满足各种复杂应用控制系统的要求。而要实现复杂控制应用的要求，接口电路就变得非常复杂，导致了应用系统的开发难度高，同时带来了开发周期长、效率低、可靠性低等弊病。为了解决这些问题，许多 IC 厂商开发了各种通用的 I/O 接口电路，把很复杂的电路集成到一片或几片大规模集成电路芯片中。这样就使得系统开发人员能够快速容易地使用这些芯片构成所需功能的 I/O 接口。为了使通用性更强，这类芯片通常可以用命令来设置其工作模式，以满足不同的控制要求。这种设置工作模式的操作通常被称为对该芯片“编程”，因此这类芯片也被称为可编程接口芯片。

在可编程接口芯片的电路中一般具有如下电路单元：

① 输入/输出数据缓冲器和锁存器，以实现数据的 I/O。

② 控制命令寄存器和状态寄存器，用以存放对外设的控制命令，以及外设的状态信息。

③ 地址译码器，用来选择接口电路中的不同端口(寄存器)。

④ 读写控制逻辑。

⑤ 中断控制逻辑。

一、片选

在微机系统中，所有 I/O 接口及内存储器都挂接在系统总线上，要访问某一接口芯片中的某个端口，必须要有地址信号选中该接口芯片才能使该接口芯片进入工作状态，访问

芯片中的某个端口。CPU 的地址信号经地址译码器后接到接口芯片的片选端 CE(ChipEnable)或 CS(ChipSelect)，CS(或 CE)究竟是高电平有效还是低电平有效视接口芯片而定。

二、读/写操作

接口芯片的地址码经译码后接通芯片的片选端，对读操作而言，使输入端口信息由数据总线进入 CPU，数据何时读入 CPU，由读信号控制。对于输出端口，CPU 对接口进行输出数据的操作时，发出写信号。在 PC 系统中，对 I/O 操作可由 IN、OUT 指令完成。

三、可编程

目前所用的接口芯片大部分是多通道、多功能的。所谓多通道就是指接口芯片同时可接几个外设；所谓多功能是指一个接口芯片能实现多种功能，实现不同的电路工作状态。而这些通道和电路工作状态的选择可由计算机指令来设定。

四、“联络”

CPU 通过外设接口芯片同外设交换信息时，接口芯片常常需要和外设有一定的“联络”信号，以保证信息的正常传输。

7.2.2. 8253 的功能及结构

在数字电路、计算机系统以及实时控制系统中常常需要用到定时信号，如函数发生器、计算机中的系统日历时钟、DRAM 的定时刷新和实时采样等。产生这些定时信号既可以用软件编程的方法，也可以用硬件的方法得到。

所谓软件定时的方法就是设计一个延时子程序，子程序中全部指令执行时间的总和就是该子程序的延时时间。在 CPU 时钟频率一定时，子程序的延时时间是固定的。这种方法比较简单，较易实现，只是需要了解延时子程序中每条指令的执行时间。软件定时的定时时间不太精确，但使用方便，因此在软件开发中经常用到。这种方法仅适用于延时时间较短、重复次数有限的场合，否则，CPU 总是执行延时程序，占用了大量的时间，使 CPU 的利用率降低。故在对时间要求严格的实时控制系统和多任务系统中很少采用。

硬件的方法是利用专用的硬件定时/计数器，在简单软件控制下产生准确的延时时间。其基本原理是通过软件确定定时/计数器的工作方式、设置计数初值并启动计数器工作，当计数到给定值时，便自动产生定时信号。这种方法的成本不高，程序上也很简单，且几乎不占用 CPU 资源，既适合长时间、多次重复的定时，也可用于延时时间较短的场合，因此得到了广泛的应用。

另外，在控制系统中还经常要对外界的某种事件进行计数，如统计传送带上的产品、工件的数量等。对这种需求，也可以利用专用的硬件定时/计数器来实现。

定时/计数器在计数方式上分为加法计数器和减法计数器。加法计数器是每有一个计数脉冲就加 1，当加到预先设定的计数值时，产生一个定时信号；减法计数器是在送入计数初值后，每来一个计数脉冲就减 1，减到 0 时产生一个定时信号输出。可编程定时/计数器 8253 是 Intel 公司专为 80×86 系列 CPU 配套使用的 16 位可编程定时/计数器芯片。

一、8253 主要功能

(1) 每片有三个独立的 16 位计数通道。

(2) 每个计数器可按二进制或十进制来计数。

(3) 每个计数器最高计数速率可达 2.6 MHz。

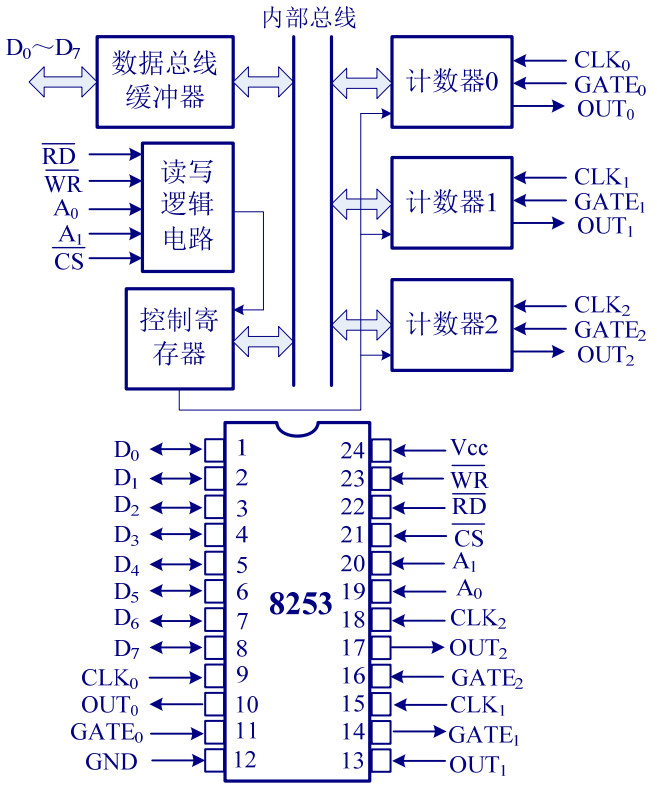
(4) 每个计数器可编程设定 6 种工作方式之一。

(5) 所有输入、输出均与 TTL 电平兼容，便于与外围接口电路相连。

二、8253 的外部引脚和内部结构

8253 的内部结构如图 7.9(a)所示。与内部总线相连的部分可为 4 部分：数据总线线缓冲器、读写控制逻辑、控制字寄存器以及三个独立的 16 位的计数器通道。这三个计数器分别是计数器 0、计数器 1 和计数器 2。

(1) 数据总线缓冲器　数据总线缓冲器是 8 位的双向三态缓冲器，主要用于 8253 与 CPU 之间进行数据传送。该数据包括三类：一是向 8253 写入的控制字，二是向计数器设置的计数初值，三是从计数器读取的计数值。



(a)内部结构图　　(b)引脚图

图 7.9　8253 内部结构及引脚图

(2) 读写控制逻辑　读写控制逻辑电路接受输入信号 $\overline{RD}$、$\overline{WR}$、$\overline{CS}$、A_1、A_0 信号，经过逻辑控制电路的组合产生相应操作，具体操作如表 7.3 所示。

表 7.3　8253 控制信号与执行的操作

$\overline{CS}$	$\overline{RD}$	$\overline{WR}$	A_1 A_0	执行的操作
0	1	0	0 0	对计数器 0 赋初值
0	1	0	0 1	对计数器 1 赋初值
0	1	0	1 0	对计数器 2 赋初值
0	1	0	1 1	写控制字
0	0	1	0 0	读计数器 0 当前计数值
0	0	1	0 1	读计数器 1 当前计数值
0	0	1	1 0	读计数器 2 当前计数值
0	0	1	1 1	无操作
1	×	×	× ×	禁止使用
0	1	1	× ×	无操作

(3) 控制字寄存器　接收 CPU 发来的对 8253 的初始化控制字。对控制字寄存器只能写入，不能读出。

(4) 三个计数器　每个计数器内部都包含一个计数初值寄存器、一个减 1 计数寄存器、一个当前计数输出寄存器和一个控制寄存器。当前计数输出寄存器跟随减 1 计数寄存器内容而变化，当有一个锁存命令出现后，当前计数输出寄存器锁定当前计数，直到被 CPU 读走之后，又随减 1 计数寄存器的变化而变化。

8253 引脚如图 7.9(b)所示，各引脚的功能定义如下：

D_7~D_0：双向，8 位三态数据总线。

CLK_0~CLK_2：输入，计数器 0、1、2 的时钟输入。

$GATE_0$~$GATE_2$：输入，计数器 0、1、2 的门控输入。

OUT_0~OUT_2：输出，计数器 0、1、2 的输出。

$\overline{CS}$：输入，片选信号，低电子有效。CPU 通过该信号有效选中 8253，对其进行读写操作。

$\overline{RD}$：输入，读信号，低电乎有效。有效时表示正读取某个计数器的当前计数值。

$\overline{WR}$：输出，写信号，低电乎有效。有效时表示正对某个计数器写入计数初值或写入控制字。

A_1、A_0：输入，片内地址选择线，可对三个计数器和控制寄存器寻址。

三、8253 的计数启动方式

8253 计数器的计数过程，可以由程序指令启动，称为软件启动；也可由外部电路信号启动，称为硬件启动。

(1)软件启动

软件启动就是 CPU 用输出指令向计数器写入初值后就启动计数。但事实上，CPU 写入的计数初值只是写到了计数器内部的初值寄存器中，计数过程并未真正开始。写入初值后的第一个 CLK 信号只是将初值寄存器中内容送到了计数器中，而从第二个 CLK 脉冲的下降沿开始，计数器才真正进行减 1 计数。之后，每来一个 CLK 脉冲都会使计数器减 1，直到减到 0 时在 OUT 端输出一个信号。因此，从 CPU 执行输出指令写入计数初值到计数结束，实际的 CLK 脉冲个数比编程写入的计数初值 N 要多一个，即 N+1 个。只要是用软件启动计数，这种误差便是不可避免的。

(2)硬件启动

硬件启动是写入计数初值后并不启动计数，而是在门控信号 GATE 由低电平变高后，再经 CLK 信号的上升沿采样，之后在该 CLK 的下降沿才开始计数。由于 GATE 信号与 CLK 信号不一定同步，故在极端情况下，从 GATE 变高到 CLK 采样之间的延时可能会经历一个 CLK 脉冲宽度，因此在计数初值与实际的 CLK 脉冲个数之间也会有一个时钟脉冲的误差。

对于大多数的工作方式，计数器每启动一次只工作一个周期(即从初值减到 0)，要想重复计数过程，则必须重新启动，这种方式称为不自动重复的计数方式。但有两种工作方式，一旦计数启动，只要门控信号 GATE 保持高电平，计数过程就会自动周而复始地重复下去，这时 OUT 端可以产生连续的波形输出，这种计数过程称为自动重复的计数方式。在自动重复计数方式下，达到稳定状态后，上面讲到的因启动造成的实际计数值和计数初值之间的误差就不再存在。

7.2.3. 8253 的工作方式

8253 共有六种不同的工作方式，在不同的工作方式下，计数过程的启动方式、OUT 端的输出波形都不一样，自动重复功能和 GATE 的控制作用以及写入新的计数初值对计数过程产生的影响也不相同。下面借助工作波形来分别说明这六种工作方式的计数过程。

一、方式 0(计数结束产生中断)

采用这种工作方式，8253 可完成计数功能，且计数器只计一遍。当控制字写入后，输出端 OUT 为低电平，当计数初值写入后，在下一个 CLK 脉冲的下降沿将计数初值寄存器内容装入减 1 计数寄存器，然后计数器开始计数，在计数期间，当计数器减为 0 之前，输出端 OUT 维持低电平。当计数值减到 0 时，OUT 输出端变为高电平，可作为中断请求信号，并保持到重新写入新的控制字或新的计数值为止。

在计数过程中，若 GATE 信号变为低电平，则在低电平期间暂停计数，减 1 计数寄存器值保持不变。

在计数过程中，若重新写入新的计数初值，则在下一个 CLK 脉冲的下降沿，减 1 计数寄存器以新的计数初值重新开始计数过程。

8253 方式 0 下三种情况的时序波形图如图 7.10 所示。

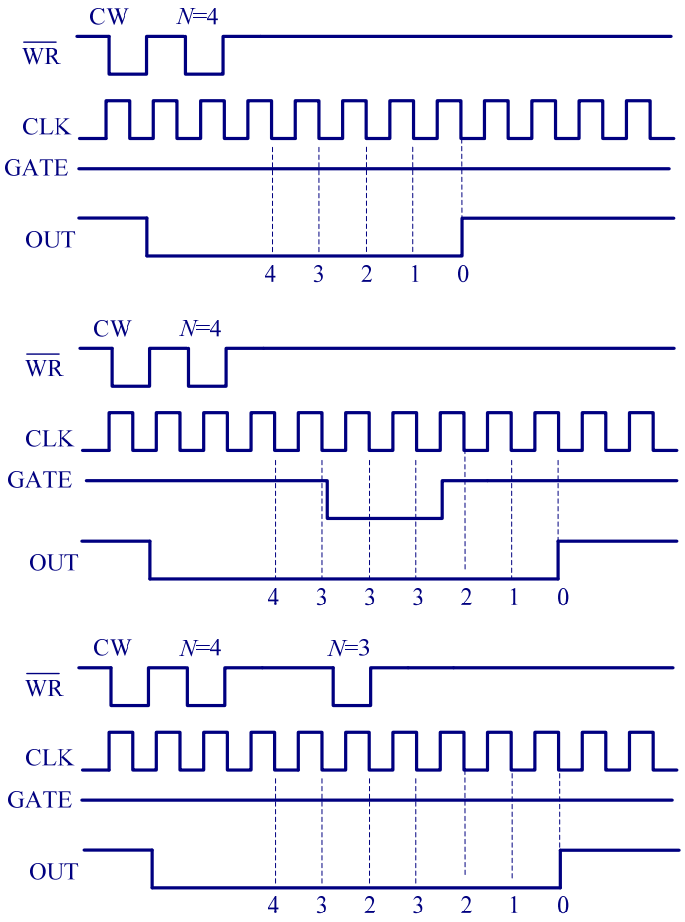


图 7.10　8253 方式 0 的时序图

二、方式 1(可重触发单稳态方式)

采用这种工作方式可输出单个负脉冲信号，脉冲的宽度可通过编程来设定。当写入控制字后，输出端 OUT 变为高电子，并保持高电平状态。然后写入计数初值，只有在 GATE 信号的上升沿之后的下一个 CLK 脉冲的下降沿，才将计数初值寄存器内容装入减 1 计数寄存器，同时 OUT 端变为低电平，然后计数器开始减 1 计数，当计数值减到 0 时，OUT 端变为高电平。

如果在 OUT 端输出低电平期间，又来一个门控信号上升沿触发，则在下一个 CLK 脉冲的下降沿，重新将计数初值寄存器内容装入减 1 计数寄存器，并开始计数，OUT 端保持低电平，直至计数值减到 0 时，OUT 端变为高电平。

在计数期间 CPU 又送来新的计数初值，不影响当前计数过程。计数器计数到 0，OUT 端输出高电平。一直等到下一次 GATE 信号的触发，才会将新的计数初值装入，并以新的计数初值开始计数过程。

8253 方式 1 下三种情况的时序波形图如图 7.11 所示。

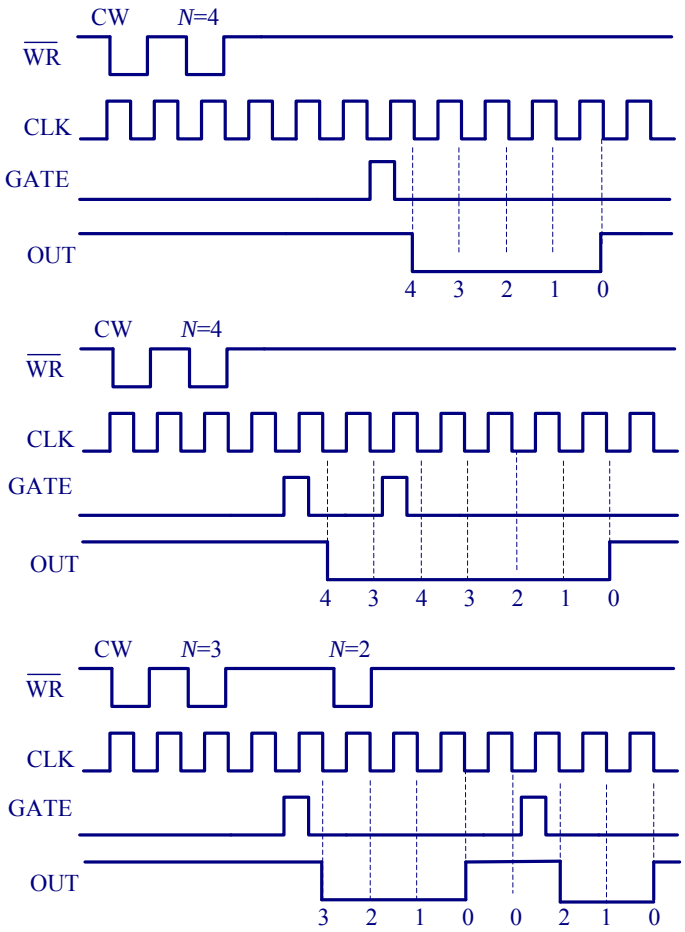


图 7.11　8253 方式 1 的时序图

三、方式 2(频率发生器)

采用方式 2，可产生连续的负脉冲信号，负脉冲宽度为一个时钟周期。写入控制字后，OUT 端变为高电平，若 GATE 为高电平，当写入计数初值后，在下一个 CLK 的下降沿将计数初值寄存器内容装入减 1 计数寄存器，并开始减 1 计数，当减 1 计数寄存器的值为 1 时，OUT 端输出低电平，经过一个 CLK 时钟周期，OUT 端输出高电子，并开始一个新的计数过程。

在减 1 计数寄存器未减到 1 时，GATE 信号由高变低，则停止计数。但当 GATE 由低变高时，则重新将计数初值寄存器内容装入减 1 计数寄存器，并重新开始计数。

GATE 信号保持高电平，但在计数过程中重新写入计数初值，则当正在计数的一轮结束并输出一个 CLK 周期的负脉冲后，将以新的初值进行计数。

8253 方式 2 下三种情况的时序波形图如图 7.12 所示。

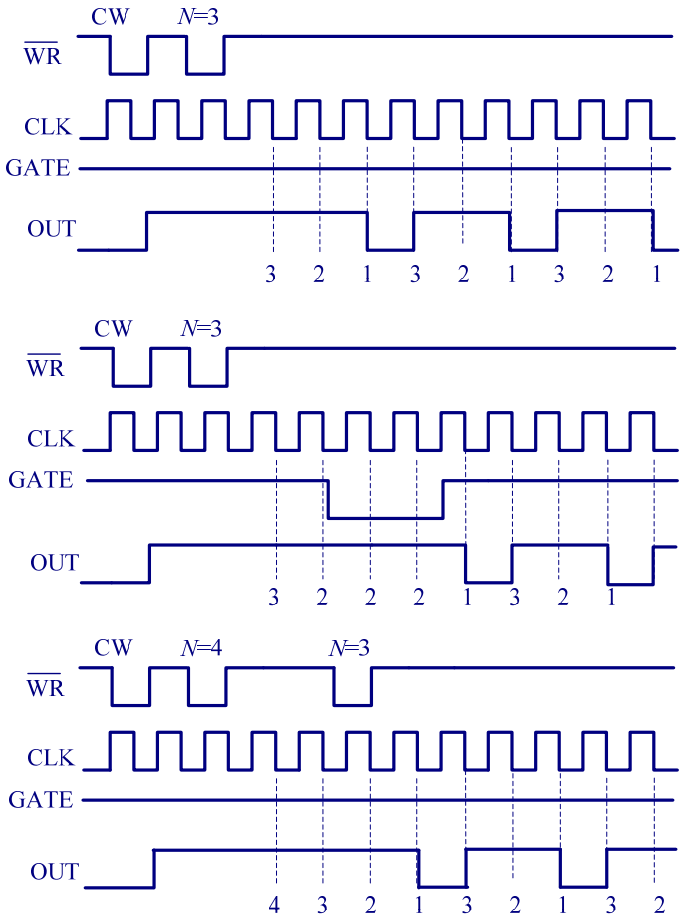


图 7.12　8253 方式 2 的时序图

四、方式 3(方波发生器)

采用方式 3，OUT 端输出方波信号。当控制字写入后，OUT 输出高电平，当写入计数初值后，在下一个 CLK 的下降沿将计数初值寄存器内容装入减 1 计数寄存器，并开始减 1 计数，当计数到一半时，OUT 端变为低电子。减 1 计数寄存器继续作减 1 计数，计数到 0 时，OUT 端变为高电平。之后，周而复始地自动进行计数过程。当计数初值为偶数时，OUT 输出对称方波；当计数初值为奇数时，OUT 输出不对称方波。

在计数过程中，若 GATE 变为低电平，则停止计数；当 GATE 由低变高时，则重新启动计数过程。如果在输出为低电平时，门控信号 GATE 变为低电平，减 1 计数器停止，而 OUT 输出立即变为高电平。在 GATE 又变成高电平后，下一个时钟脉冲的下降沿，减 1 计数器重新得到计数初值，又开始新的减 1 计数。

在计数过程中，如果写入新的计数值，那么，将不影响向当前输出周期。但是，如果在写入新的计数值后，又受到门控上升沿的触发，那么，就会结束当前输出周期，而在下一个时钟脉冲的下降沿，减 1 计数器重新得到计数初值，又开始新的减 1 计数。

8253 方式 3 下三种情况的时序波形图如图 7.13 所示。

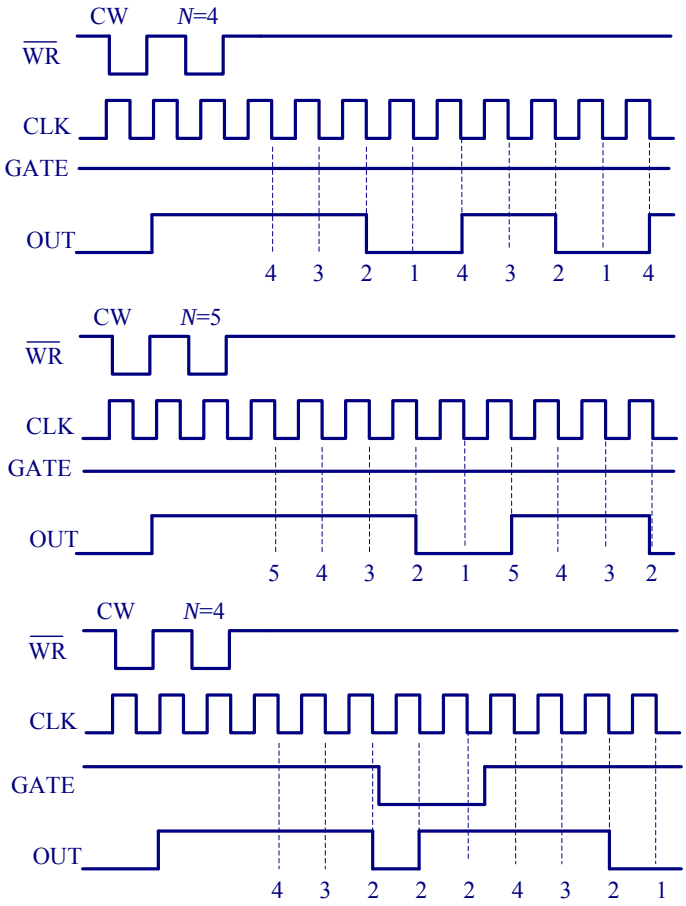


图 7.13　8253 方式 3 的时序图

五、方式 4(软件触发的选通信号发生器)

采用方式 4，可产生单个负脉冲信号，负脉冲宽度为一个时钟周期。写入控制字后，OUT 端变为高电平，若 GATE 为高电平，当写入计数初值后，在下一个 CLK 的下降沿将计数初值寄存器内容装入减 1 计数寄存器，并开始减 1 计数，当减 1 计数寄存器的值为 0 时，OUT 端输出低电平，经过一个 CLK 时钟周期，OUT 端输出高电平。

如果在计数时，又写入新的计数值，则在下一个 CLK 的下降沿此计数初值被写入减 1 计数寄存器，并以新的计数值作减 1 计数。

8253 方式 4 下三种情况的时序波形图如图 7.14 所示：

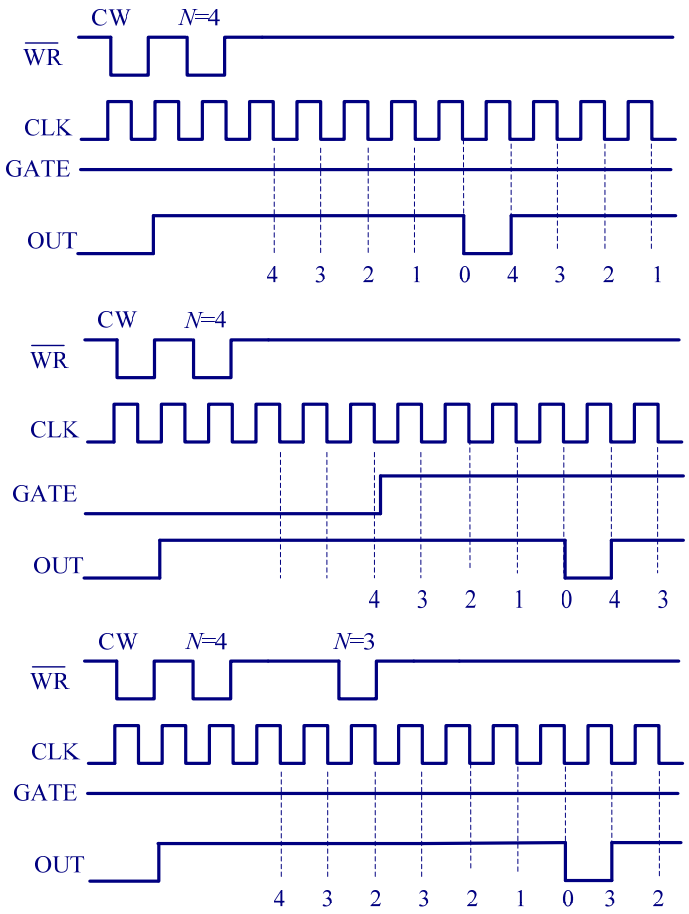


图 7.14　8253 方式 4 的时序图

六、方式 5(硬件触发的选通信号发生器)

方式 5 的计数过程由 GATE 的上升沿触发。当控制字写入后，OUT 端输出高电平，并保持高电平状态。然后写入计数初值，只有在 GATE 信号的上升沿之后的下一个 CLK 脉冲的下降沿，才将计数初值寄存器内容装入减 1 计数寄存并开始减 1 计数，当计数值减到 0 时，OUT 端变为低电平，并持续一个 CLK 周期，然后自动变为高电平。

若在计数过程中，GATE 端又来一个上升沿触发，则在下一个 CLK 脉冲的下粒沿，减 1 计数寄存器将重新获得计数初值，并按新的初值作减 1 计数，直至减至 0 为止。若在计数过程中，写入新的计数值，但没有触发脉冲，则当前输出周期不受影响。当前周期结束后，在再触发的情况下，将按新的计数初值开始计数。若在计数过程中，写入新的计数值，并在当前周期结束前又受到触发，则在下一个 CLK 脉冲的下降沿，减 1 计数寄存器将获得新的计数初值，并按此值作减 1 计数操作。

8253 方式 5 下三种情况的时序波形图如图 7.15 所示。

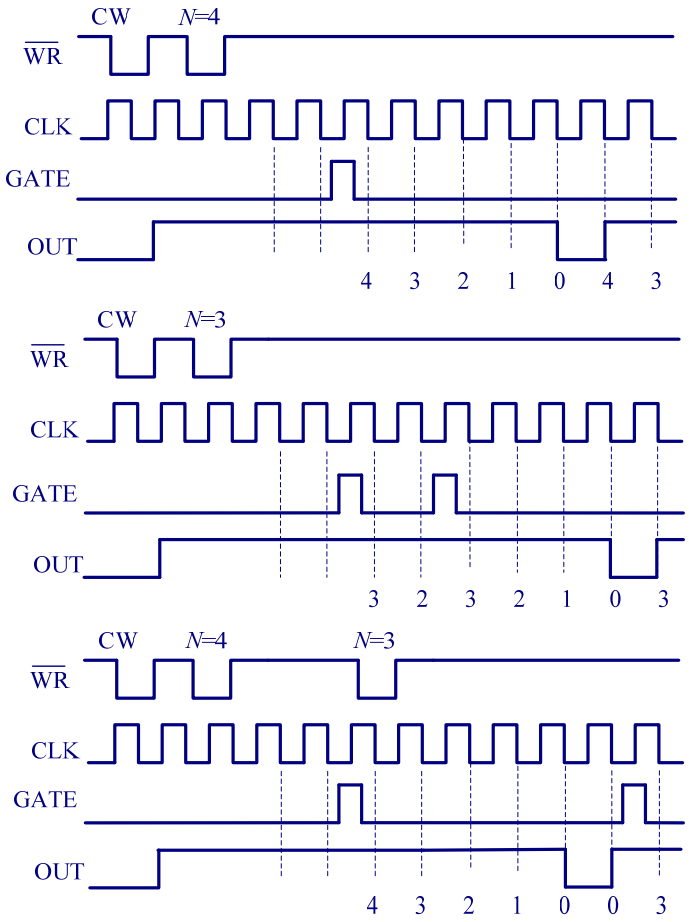


图 7.15　8253 方式 5 的时序图

对 8253 的 6 种工作方式，在操作时应遵守以下三条基本原则：

(1) 当控制字写入 8253 时，所有的控制逻辑电路自动复位，这时输出端 OUT 进入初始态。

(2) 当初始值写入计数器后，要经过一个时钟周期，减法计数器才开始工作，时钟脉冲的下降沿使计数器进行减 1 操作。计数器的最大初始值是 0，用二进制计数时 0 相当于 2^{16}，用十进制计数时 0 相当于 10^4。

(3) 一般情况下，在时钟脉冲 CLK 的上升沿采样门控信号。门控信号的触发方式有边沿触发和电平触发两种。门控信号为电平触发的有：方式 0，方式 4。门控信号为上升沿触发的有：方式 1，方式 5。门控信号可为电平触发也可为上升沿触发的有：方式 2，方式 3。

表 7.4 将 8253 计数器六种工作方式的特点综合在一起，以便于读者比较。

表 7.4　8253 计数器工作方式一览表

方式	启动计数	中止计数	自动重复	更新初值	输出波形
0	软件(写入	GATE=0	否	立 即 有	延 时 时 间 可

	初值)			效	变的上跳沿
1	硬　件(GATE 正跳变)		否	下一轮有效	宽度为 $N\times T_{CLK}$ 的单一负脉冲
2	软件(写入初值)；硬件(GATE 正跳变)	GATE=0	是	下一轮有效	周期为 $N\times T_{CLK}$、宽度为 T_{CLK} 的连续负脉冲
3	软件(写入初值)；硬件(GATE 正跳变)	GATE=0	是	下半轮有效	周期为 $N\times T_{CLK}$ 的连续方波
4	软件(写入初值)	GATE=0	否	下一轮有效	宽度为 T_{CLK} 的单一负脉冲
5	硬　件(GATE 正跳变)		否	下一轮有效	宽度为 T_{CLK} 的单一负脉冲

7.2.4. 8253 的控制字及初始化编程

为让 8253 计数器工作，必须先设置控制寄存器的控制字，用来选择计数器、设置工作方式、计数方法以及 CPU 访问计数器的读写方法等。8253 控制字为 8 位，其格式如图 7.16 所示，其中 D_7D_6 用于选择定时器；D_5D_4 用于确定时间常数的读/写格式；$D_3D_2D_1$ 用来设定计数器的工作方式；D_0 用来设定计数方式。

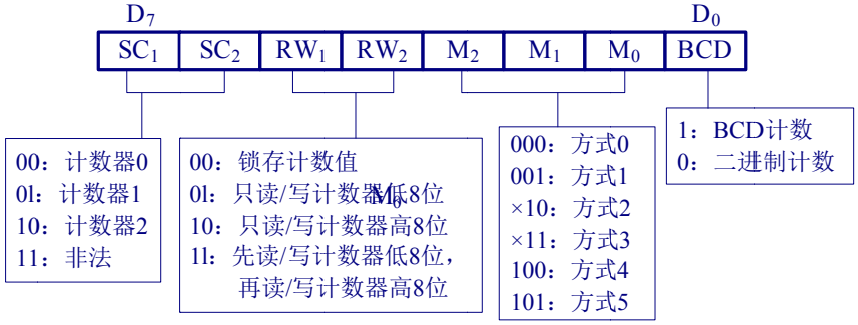


图 7.16　8253 控制字格式

8253 的控制寄存器和三个计数器分别具有独立的编程地址，由控制字的内容确定使用的是哪个寄存器以及执行什么操作。因此 8253 在初始化编程时并最有严格的顺序规定，但在编程时，必须遵守两条原则：①在对某个计数器设置初值之前，必须先写入控制字。②在设计初始值时，要符合控制字中规定的格式，即只写低位字节，还是只写高位字节，或高、低位字节都写(分两次写，先低字节后高字节)。

8253 编程命令有两类：一类是写入命令，包括设置控制命令字、设置计数器的初始值

命令和锁存命令。另一类是读出命令，用来读取计数器的当前值。锁存命令是配合读出命令使用的，在读计数值前，必须先用锁存命令锁定当前计数输出寄存器的当前计数。否则，在读数时，减 1 计数寄存器的值处在动态变化过程中，当前计数输出寄存器随之变化，就会得到一个不确定的结果。当 CPU 将此锁定值读走之后，锁存功能自动失锁，于是当前计数输出寄存器的内容又跟随减 1 计数寄存器而变化。在锁存和读出计数值的过程中，减 1 计数寄存器仍在作正常减 1 计数，这样，保证了计数器在运行中被读取而不影响计数的进行。

7.2.5. 应用实例

例 7.4　某 8086 微机系统中，8253 的三个计数器端口地址分别为 3F0H，3F1H，3F2H，控制字寄存器端口地址为 3F3H，要求通道 0 工作于方式 3，且计数初值 *TC*=1234 。则初始化程序为：

```
MOV AL, 00110111B        ;控制字
MOV DX, 3F3H             ; 控制端口
OUT DX, AL               ;送控制字
MOV DX, 3F0H             ;通道 0 口的地址
MOV AL, 34H              ;计数值低字节
OUT DX, AL               ;写低字节
MOV AL, 12H              ;计数值高字节
OUT DX, AL               ;写高字节
```

读当前计数值的程序为：

```
MOV AL, 00000111B        ;控制字
MOV DX, 3F3H             ;控制端口
OUT DX, AL               ;送控制字
MOV DX, 3F0H             ;通道 0 口的地址
IN AL, DX                ;读低字节
MOV AH, AL               ;保存
IN AL, DX                ;读高字节
XCHG AH, AL              ;存入 AX
MOV CX, 1234+1           ;求计数值
SUB CX, AX               ;得到计数值
```

例题 7.5　8253 在 IBM PC 中的应用。IBM PC 系统板上 8253 的接口电路如图 7.16 所示，三个计数器的时钟输入频率为 1.193 2 MHz。系统分配给 8253 的端口地址为 40H~43H。

计数器 0 为方式 3，先写低字节，后写高字节，二进制计数，计数初值为 0。输出端 OUT0

接至中断控制器 8259A 的 IR0，OUT0 输出的脉冲周期约为 55 ms(65536÷1193200)，即计数器 0 每隔 55 ms 产生一次中断请求。

计数器 1 为方式 2，只写低字节，二进制计数，计数初值为 18。输出端 OUT_1 接至 DMA 控制器 8237A 通道 0 的 DMA 请求 DREQ。，作为定时(15.08 μs)刷新动态存储器的启动信号。

计数器 2 为方式 3，控制扬声器发出频率为 1 kHz 的声音，故取计数初值为 1190。$GATE_2$ 由 8255A 的 PB0 控制，当 $GATE_2$ 为高电平时，OUT_2 输出频率为 1kHz 的方波，经功率放大器和滤波后驱动扬声器发声。在 IBM PC 中，要使扬声器发声，还必须使 8255 的 PB1 输出高电平。8255 的 B 口地址为 61H。

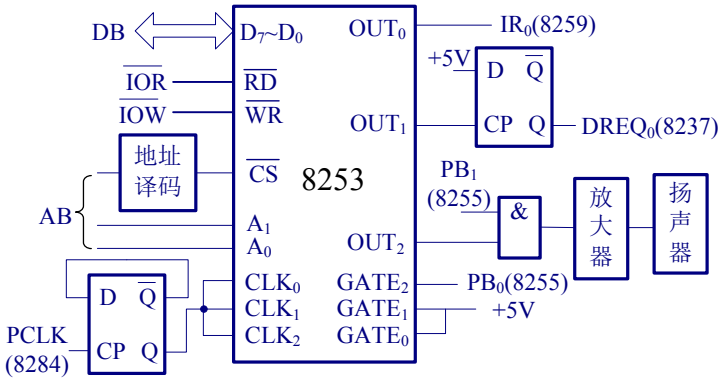


图 7.16　IBM PC 系统板上 8253 的接口电路

满足上述要求的 8086 程序如下：

```
;CNT0 初始化
MOV AL, 36H     ;选择计数器 0，写双字节计数值，方式 3，二进制计数
OUT 43H, AL     ;控制字写入控制寄存器
MOV AL, 0       ;选最大计数值(65 536)
OUT 40H, AL     ;写低 8 位计数值
OUT 40H, AL     ;写高 8 位计数值
;CNT1 初始化
MOV AL, 54H     ;选择计数器 1，低 8 位单字节计数值，方式 2，二进制计数
OUT 43H, AL
MOV AL, 18
OUT 41H, AL     ;计数值写入计数器 1
;CNT2 初始化
MOV AL, 0B6H    ;选择计数器 2，双字节计数值，方式 3，二进制计数
OUT 43H, AL
```

```
MOV AX, 1190
OUT 42H, AL      ;送低字节到计数器 2
MOV AL, AH       ;(AH)←高字节计数值
OUT 42H, AL      ;高 8 位计数值写入计数器 2
IN AL, 61H       ;读 8255 的 B 口
MOV AH, AL       ;将 B 口内容保存
OR AL, 03        ;使 PB0=PBl=1
OUT 61H, AL      ;使扬声器发声
    …
MOV AL, AH       ;恢复 8255B 口状态
OUT 61H, AL
```

例 7.6　脉冲发生器。电路如图 7.17 所示。3 个计数器 CLK 频率均为 2MHz。要求计数器 0 在定时 100μs 后产生中断请求；计数器 1 用于产生周期为 10μs 的对称方波；计数器 2 每 1ms 产生一个负脉冲。编写 8253 的初始化程序。

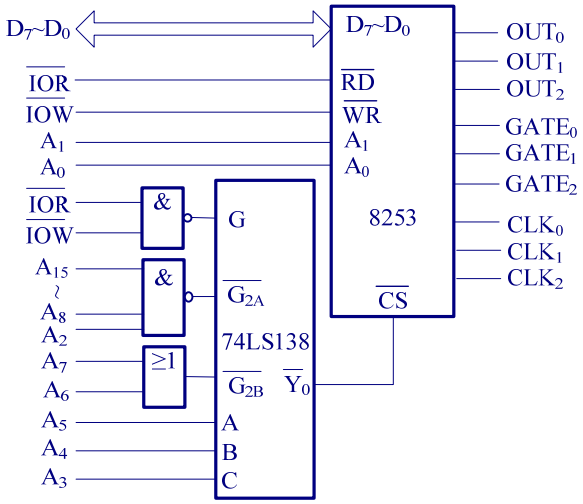


图 7.17　脉冲发生器

解　根据要求可知，计数器 0 应工作于方式 0，计数初值=100μs/0.5μs=200(CLK 的周期=0.5μs)；计数器 1 应工作于方式 3，计数初值=10μs/0.5μs=20；计数器 2 应工作于方式 2，计数初值=1 ms/0.5μs=2000。以下是 8253 的初始化程序。

```
INIT8253: MOV DX, 0FF07H
  MOV AL, 10H      ；计数器 0，只写计数值低 8 位，方式 0，二进制计数
  OUT DX, AL
```

```
MOV AL, 56H       ；计数器 1，只写计数值低 8 位，方式 3，二进制计数
OUT DX, AL
MOV AL, 0B4H      ；计数器 2，先写高 8 位，再写低 8 位，方式 2，二进制计数
OUT DX, AL
MOV DX, 0FF04H
MOV AL, 200       ；计数器 0 的计数初值
OUT DX, AL
MOV DX, 0FF05H
MOV AL, 20        ；计数器 1 的计数初值
OUT DX, AL
MOV DX, 0FF06H
MOV AX, 2000      ；计数器 2 的计数初值
OUT DX, AL
MOV AL, AH
OUT DX, AL
```

本例中，8253 通过对外部输入时钟信号的计数，可以达到计数和定时两种应用目的。门控信号 CATE 提供了从外部控制计数器的能力。当一个计数器计数或定时长度不够时，还可以把两个、三个计数器串联起来使用，即一个计数器的输出 OUT 作为下一个计数器的外部时钟 CLK 输入，甚至可将两个 8253 串起来使用。

例 7.7　利用 8253 的通道 0 和通道 1，设计并产生周期为 1 Hz 的方波。设通道 0 的输入时钟频率为 2 MHz，8253 的端口地址为 80H，81H，82H，83H。

解　根据题意可知通道 0 的输入时钟周期为 0.5μs，其最大定时时间为 0.5μs×65536，即为 32.768 ms，要产生频率为 1 Hz(周期为 1 s)的方波，单独利用一个通道是无法实现的。但可利用通道级联的方法，将通道 0 的输出 OUT_0 作为通道 1 的输入时钟。若让 8253 通道 0 工作于方式 2(速率发生器)，输出脉冲周期为 10 ms，则通道 0 的计数值为 20 000。周期为 10 ms 的脉冲作为通道 1 的输入，要求输出端 OUT_1 的波形为方波且周期为 1s，则通道 1 的计数值为 100。

通过以上分析，硬件连接图如图 7.18 所示。

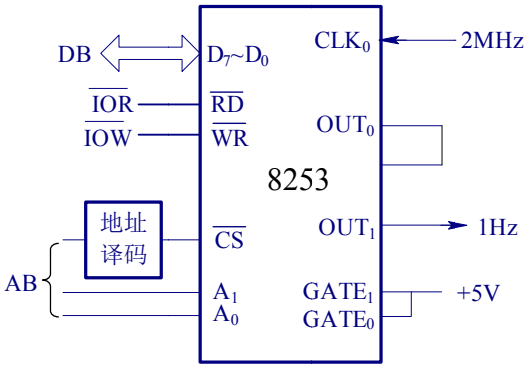


图 7.18　方波发生电路

8253 初始化程序如下：

```
MOV AL, 34H       ;通道 0 控制字
OUT 83H, AL
MOV AX, 20000     ;通道 0 时间常数
OUT 80H, AL
MOV AL, AH
OUT 80H, AL
MOV AL, 56H       ;通道 1 控制字
OUT 83H, AL
MOV AL, 100       ;通道 1 时间常数
OUT 81H, AL
```

7.3. 可编程外围接口芯片 8255A 及其应用

7.3.1. 8255A 的功能及结构

并行接口一次可以同时传送一个数据的所有位。对一个具体的并行接口来说，其数据传送方向有两种，一是单向传送(只作为输入口或输出口)，另一种是双向传送(既可作为输入口，也可作为输出口)。并行接口可以很简单(如锁存器或三态门)，也可以很复杂(如可编程并行接口芯片)，功能完善的并行接口中一般都包括输入／输出数据寄存器、控制寄存器(存放控制命令)、状态寄存器(保存当前工作状态)和总线缓冲器等部件。

8255A 是 Intel 公司为 80×86 系列 CPU 配套的可编程并行接口芯片。8255A 的通用性较强，使用灵活，是一种典型的可编程并行接口。

一、8255A 内部结构

8255A 的内部结构如图 7.19(a)所示，它由 4 部分组成：

(1) 数据总线缓冲器

数据总线缓冲器是一个双向三态的 8 位数据缓冲器，8255A 通过它与系统总线相连。输入数据、输出数据、CPU 发给 8255A 的控制字都是通过这个缓冲器进行的。

(2) 数据端口 A、B、C

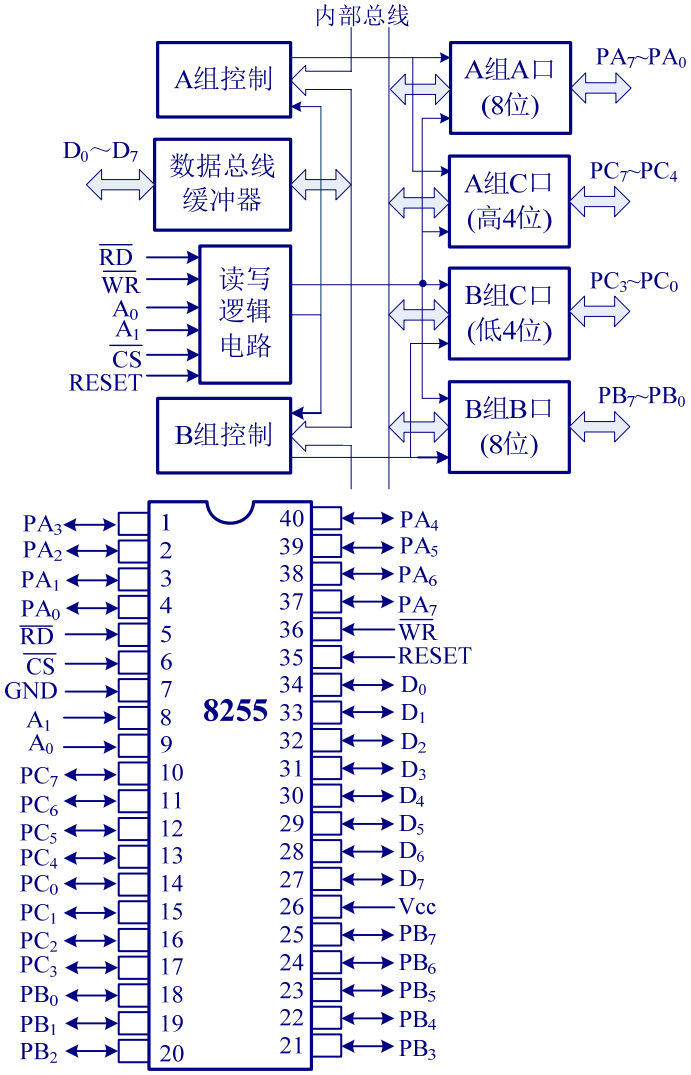
8255A 由三个 8 位数据端口，即端口 A、端口 B、端口 C。设计人员可通过编程使它们分别作为输入端口或输出端口。不过，这三个端口有各自的特点。

端口 A 对应一个 8 位数据输入锁存器和一个 8 位数据输出锁存器/缓冲器。用端口 A 作为输入或输出时，数据均受到锁存。

端口 B 和端口 C 均对应一个 8 位输入缓冲器和一个 8 位数据输出锁存器/缓冲器。

在使用中，端口 A 和端口 B 常常作为独立的输入或者输出端口。端口 C 除了可以做独立的输入或输出端口外，还可配合端口 A 和端口 B 的工作。具体说，端口 C 可分成两个 4 位的端口，分别作为端口 A 和端口 B 的控制信号和状态信号。

(3)A 组控制和 B 组控制



(a)内部结构图　　(b)引脚图

图 7.19　8255A 内部结构及引脚图

这两组控制电路一方面接收 CPU 发来的控制字并决定 8255A 的工作方式；另一方面接收来自读/写控制逻辑电路的读／写命令，完成接口的读／写操作。

A 组控制电路控制端口 A 和端口 C 的高 4 位的工作方式和读／写操作。

B 组控制电路控制端口 B 和端口 C 的低 4 位的工作方式和读／写操作。

(4)读/写控制逻辑

读/写控制逻辑负责管理 8255A 的数据传输过程。它接收译码电路的$\overline{CS}$和来自地址总线的 A_1、A_0 信号，以及控制总线的 RESET、$\overline{RD}$、$\overline{WR}$ 信号，将这些信号进行组合后，得到对 A 组控制部件和 B 组控制部件的控制命令，并将命令发给这两个部件，以完成对数据信息、状态信息和控制信息的传输。

二、8255A 引脚功能

8255A 芯片除电源和地引脚以外，其他引脚可分为两组，引脚如图 7.19(b)所示：

(1) 8255A 与外设连接引脚

8255A 与外设连接的有 24 条双向、三态数据引脚，分成三组，分别对应于 A、B、C 三个数据端口：PA_7～PA_0、PB_7～PB_0、PC_7～PC_0。

(2) 8255 与 CPU 连接引脚

D_7～D_0：双向、三态数据线。

RESET：复位信号，高电平有效。复位时所有内部寄存器清除，同时 3 个数据端口被设为输入。

$\overline{CS}$：片选信号，低电平有效。该信号有效时，8255A 被选中。

$\overline{RD}$：读信号，低电平有效。该信号有效时，CPU 可从 8255A 读取输入数据或状态信息。

$\overline{WR}$：写信号，低电平有效。该信号有效时，CPU 可向 8255A 写入控制字或输出数据。

A_1、A_0：片内端口选择信号。8255A 内部有三个数据端口和一个控制端口。

8255A 的$\overline{CS}$、$\overline{RD}$、$\overline{WR}$、A_1、A_0 控制信号和传送操作之间的关系表 7.5 所示。

表 7.5　8255A 的控制信号和传送操作的对应关系

$\overline{CS}$	$\overline{RD}$	$\overline{WR}$	A_1　A_0	执行的操作
0	0	1	0　0	读 A 口

0	0	1	0	1	读 B 口
0	0	1	1	0	读 C 口
0	0	1	1	1	非法状态
0	1	0	0	0	写 A 口
0	1	0	0	1	写 B 口
0	1	0	1	0	写 C 口
0	1	0	1	1	写控制寄存器
1	×	×	×	×	未选通

图 7.20 给出了 8255 各引出线与系统总线的连接示意图。

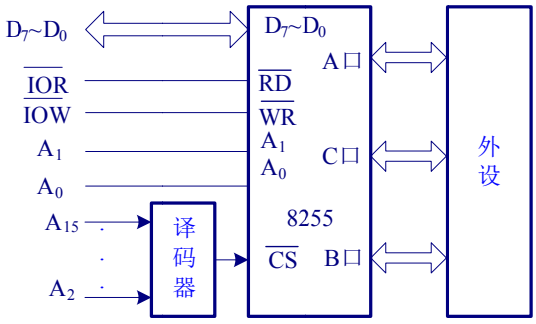


图 7.20　8255 与系统总线的连接示意图

7.3.2. 8255A 的工作方式

一、方式 0——基本输入输出

方式 0 下，每一个端口都作为基本的输入或输出口，端口 C 的高 4 位和低 4 位以及端口 A、端口 B 都可独立地设置为输入口或输出口，如图所示。4 个端口的输入/输出可有 16 种组合。

8255A 工作于方式 0 时，CPU 可采用无条件读写方式与 8255A 交换数据，也可采用查询方式与 8255A 交换数据。采用查询方式时，可利用端口 C 作为与外设的联络信号，如图 7.21 所示。

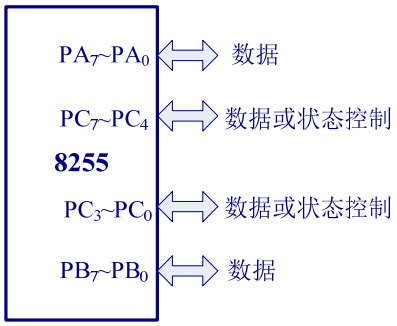


图 7.21　8255A 工作于方式 0

二、方式 1——选通输入输出

这种方式也称为选通的输入/输出方式。在这种工作方式下，A、B、C 3 个口被分为两组。A 组包括 A 口和 C 口的高 4 位，A 口可由编程任意设定为输入口或输出口，C 口的高 4 位则用来作为 A 口输入/输出操作的控制和同步信号；B 组包括 B 口和 C 口的低 4 位，B 口可由编程任意设定为输入口或输出口，C 口的低 4 位则用来作为 B 口输入/输出操作的控制和同步信号，如图 7.22 所示。在方式 1 下 A 口和 B 口的输入数据和输出数据都被锁存。

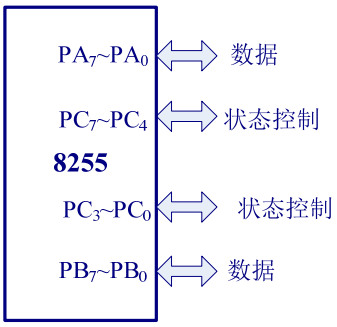


图 7.22　8255A 工作于方式 1

(1) 方式 1 输入

端口 A、端口 B 都设置为方式 1 输入时的情况如图 7.23 所示，在这种情况下 PC_3、PC_4、PC_5 作为端口 A 的联络信号，PC_0、PC_1、PC_2 作为端口 B 的联络信号。

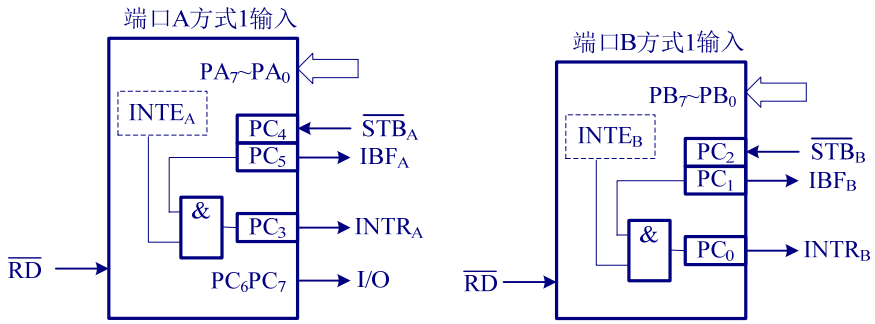


图 7.23　8255A 端口 A 和 B 都设置为方式 1 输入

各联络信号的含义如下：

$\overline{STB}$：选通输入，低电平有效。该信号有效时，输入数据被送入锁存端口 A 或端口 B 的输入锁存器 / 缓冲器中。

IBF：输入缓冲器满，高电平有效。该信号由 8255A 发出，作为$\overline{STB}$信号的应答信号。该信号有效时，表明输入缓冲器中已存放数据，可供 CPU 读取。IBF 由$\overline{STB}$信号为 0 时置位，由$\overline{RD}$信号的上升沿复位。

INTR：中断请求信号，高电平有效。当 IBF 和 INTE 均为高电平时，INTR 变 为高电平。INTR 信号可作为 CPU 的查询信号，或作为向 CPU 发出中断请求的 $\overline{RD}$信号的下降沿使 INTR 复位，上升沿又使 IBF 复位。

INTE：中断允许信号。端口 A 用 PC_4 的置位/复位控制，端口 B 用 PC_2 的置位/复位控制。需特别说明的是，对 INTE 信号的设置，虽然使用的是对端口 C 的置位/复位操作，但这完全是 8255A 的内部操作，对已作为$\overline{STB}$信号的引脚 PC_4、PC_2 的逻辑状态没有影响。

方式 1 下数据输入过程为：当外设要发送数据时，就将数据送到 8255 的 A 口或 B 口上，并利用$\overline{STB}$脉冲将数据锁存到 8255 的输人数据锁存器中。$\overline{STB}$还会使 IBF 有效并产生 INTR 信号，有效的 IBF 通知外设数据已被锁存，INTR 信号则可用于通过 8259 中断控制器向 CPU 提出中断请求，要求 CPU 从 8255 的输入端口上读取数据。CPU 响应中断并读取数据后使 IBF 和 INTR 变为无效。

8255A 工作在方式 1 下的输入时序如图 7.24 所示。

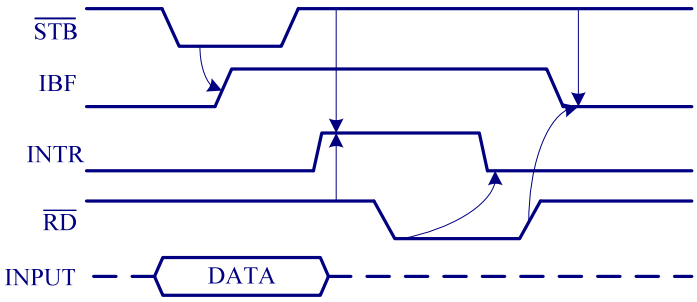


图 7.24　方式 1 输入的时序

(2) 方式 1 输出

端口 A、端口 B 都设置为方式 1 输出时的情况如图 7.25 所示，在这种情况下 PC_3、PC_6、PC_7 作为端口 A 的联络信号，PC_0、PC_1、PC_2 作为端口 B 的联络信号。

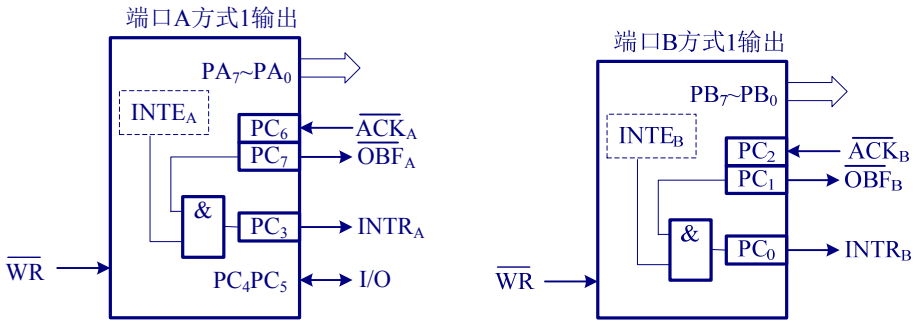


图 7.25　8255A 端口 A 和 B 都设置为方式 1 输出

各联络信号的含义如下：

$\overline{OBF}$：输出缓冲器满，低电平有效。该信号有效时，表明 CPU 已将待输出的数据写入到 8255A 的指定端口，通知外设可从指定端口读取数据。该信号由 $\overline{WR}$ 的上升沿置为有效。

$\overline{ACK}$：响应信号，低电平有效。该信号由外设发给 8255A，有效时，表示外设已取走 8255A 的端口数据。

INTR：中断请求信号，高电平有效。当输出缓冲器空($\overline{OBF}$ =1)，中断允许 INTE=1 时，INTR 变为高电平。INTR 信号可作为 CPU 的查询信号，或作为向 CPU 发出中断请求的信号。 $\overline{WR}$ 的下降沿使 INTR 复位。

INTE：中断允许信号。端口 A 用 PC_6 的置位/复位控制，端口 B 用 PC_2 的置位/复位控制。

INTR 是否输出高电平是由 INTE 和 $\overline{ACK}$ 信号共同决定的。以 A 口为例，当 CPU 向接口写数据时(执行一条 OUT 指令)，在 $\overline{IOW}$ 有效期间将数据锁存于芯片的数据缓冲器中，之后在 $\overline{IOW}$ 的上升沿使 $\overline{OBF}$ =0(PC_7 端输出负脉冲)，通知外部设备，A 口已有数据准备好。一旦外设将数据取走，它就送出一个有效的 $\overline{ACK}$ 脉冲，该脉冲使 $\overline{OBF}$ =1，若 INTE 也为高电平，就会在 PC_3 端产生一个有效的 INTR 信号。该信号可接到中断控制器 8259 的中断请求线 IR 端，进而向 CPU 提出中断请求。CPU 响应中断后，向接口写入下一个数据，同样由 $\overline{IOW}$ 将数据锁存，当数据锁存并由信号线输出，8255 就去掉 INTR 信号并使 $\overline{OBF}$ 有效，重复上述过程。

8255A 工作在方式 1 下的输出时序如图 7.26 所示。

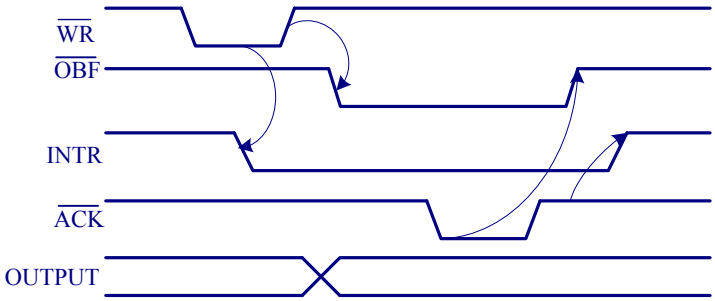


图 7.26　方式 1 输出的时序

三、方式 2——双向选通输入输出

方式 2 为双向传输方式。8255A 的方式 2 可使 8255A 与外设进行双向通信，既能发送数据，又能接收数据。可采用查询方式和中断方式进行传输。

方式 2 只适用于端口 A，端口 C 的 PC_7～PC_3 配合端口 A 的传输，其联络信号如图 7.27 所示。INTEl 为输出中断允许，由 PC_6 置位/复位；INTE2 为输入中断允许，由 PC_4 置位/复位。

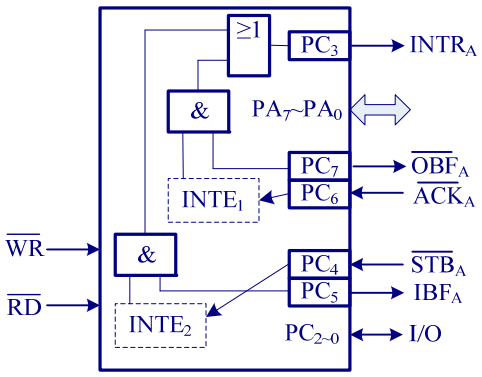


图 7.27　工作方式 2 的信号定义

当端口 A 工作于方式 2，端口 B 工作于方式 0 时 PC_7～PC_3 作为端口 A 的联络信号，PC_0～PC_2 可工作于方式 0；当端口 A 工作于方式 2，端口 B 工作于方式 1 时，PC_7～PC_3 作为端口 A 的联络信号，PC_0～PC_2 作为端口 B 的联络信号。

8255A 工作在方式 2 下的输出时序如图 7.28 所示。此时的 A 口可以认为是方式 1 的输入和输出的组合(当然，A 口不可能在某一个时刻既输出又输入，而是在某一时刻输出，另一时刻输入。输出/输入操作是分时进行的)。实际传输过程中，输入和输出的顺序以及各自操作的次数是任意的，只要 $\overline{WR}$ 在 $\overline{ACK}$ 之前发出，$\overline{STB}$ 在 $\overline{RD}$ 之前发出就可以了。

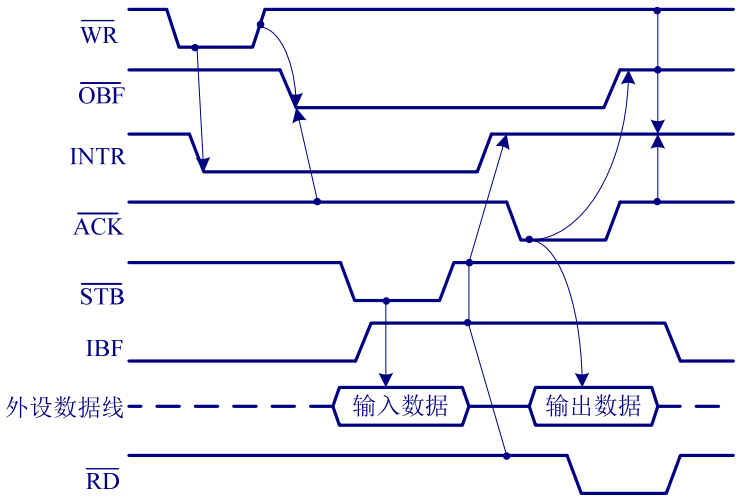


图 7.28　8255A 方式 2 的控制信号和时序

在输出时，CPU 发出写脉冲 $\overline{WR}$，向 A 口写人数据。$\overline{WR}$ 信号使 INTR 变低电平，同时使 $\overline{OBF}$ 有效。外设接到面信号后发出 $\overline{ACK}$ 信号，从 A 口读出数据。$\overline{ACK}$ 信号使 $\overline{OBF}$ 无效，并使 INTR 变高电平(如图 8．31 所示的虚线部分)，产生中断请求，准备输出下一个数据。

输入时，外设向 8255 送来数据，同时发 $\overline{STB}$ 信号给 8255，该信号将数据锁存到 8255

的 A 口，从而使 IBF 有效。$\overline{STB}$ 信号结束使 INTR 有效，向 CPU 请求中断。CPU 响应中断后，发出读信号 $\overline{RD}$，从 A 口中将数据读走。$\overline{RD}$ 信号会使 INTR 和 IBF 信号无效，从而开始下一个数据的读入过程。

值得注意的是，在工作方式 2 下，8255 与外设之间是通过 A 口的 8 根线 PA_0~PA_7 交换数据的。在 PA_0~PA_7 上，随时可能出现输出到外设的数据，也可能出现外设送给 8255 的数据，这就要防止 CPU 和外设同时竞争 PA_0~PA_7 数据线。

7.3.3. 8255A 的控制字和状态字

8255A 有两个控制字：方式选择控制字和端口 C 置位/复位控制字。这两个控制字共用一个地址，即控制端口地址。用控制字的 D_7 位来区分这两个控制字，D_7=1 为方式选择控制字；D_7=0 为端口 C 置位/复位控制字。

一、方式选择控制字

方式选择控制字的格式如图 7.29 所示，其中 D_0~D_2 用来对 B 组的端口进行工作方式设定，D_3~D_6 用来对 A 组的端口进行工作方式设定。最高位为 1 是方式选择控制字标志。

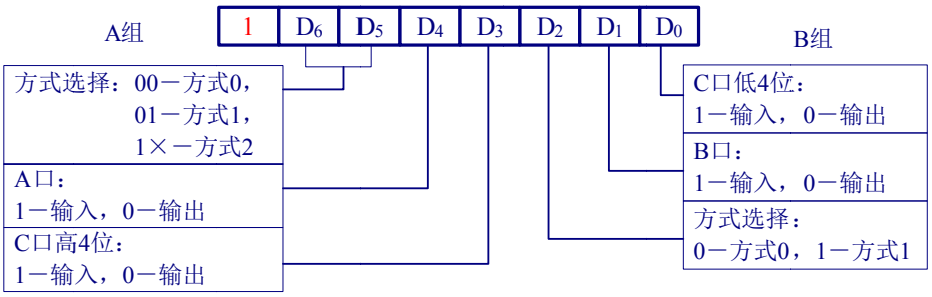


图 7.29　8255A 方式选择控制字

二、端口 C 置位/复位控制字

端口 C 置位/复位控制字的格式如图 7.30 所示，其中 D_3~D_1 三位的编码与端口 C 的某一位相对应，D_0 决定置位或复位操作。最高位为 0 是端口 C 置位/复位控制字标志。

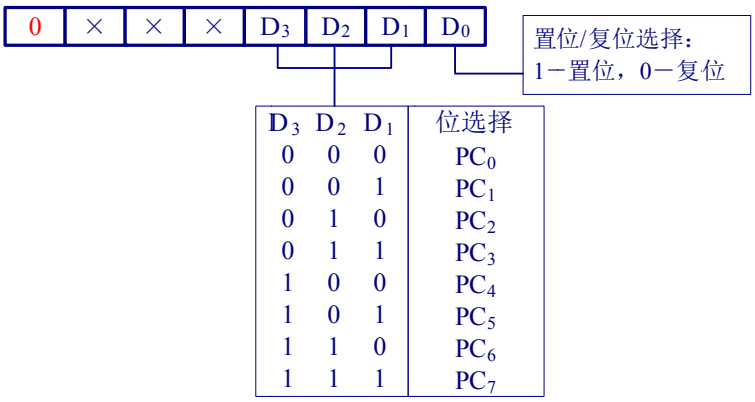
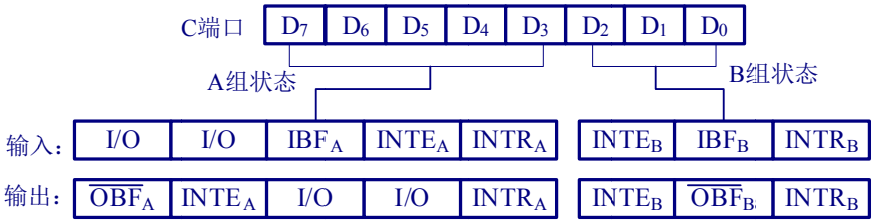


图 7.30　8255A 端口 C 置位/复位控制字

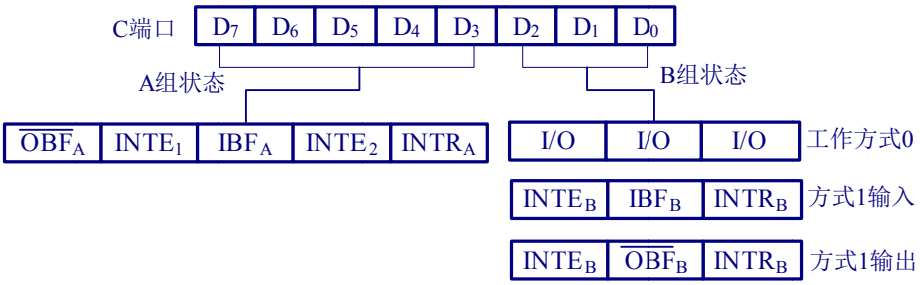
三、状态字

状态字反映了 C 端口各位当前的状态。当 8255A 的 A 口、B 口工作在方式 1 或 A 口工作在方式 2 时，通过读 C 口的状态，可以检测 A 口和 B 口当前的工作情况。A、B 口工作在不同方式下的状态字各位的含意分别如图 7.31 所示。方式 1 下的状态字如图 7.31(a)所示，方式 2 下的状态字如图 7.31(b)所示。需要说明的是，图 7.31(a)表示在方式 1 下，A 口、B 口同为输入或同为输出的情况。若在此方式下，A 口、B 口的输入／输出方式不同时，状态字为上述两状态字的组合。



(a)

方式 2 下的状态字



(b)

图 7.31　A、B 口工作在不同方式下的状态字各位的含意

7.3.4. 应用实例

例 7.8　采用 8255A 的接口电路如图 7.32 所示，其中 8255A 的 A 口接 8 个开关，B 口接 8 个发光二极管。试编程点亮发光二极管，使之与开关的接通状态一致，当开关合上时对应的发光二极管点亮。

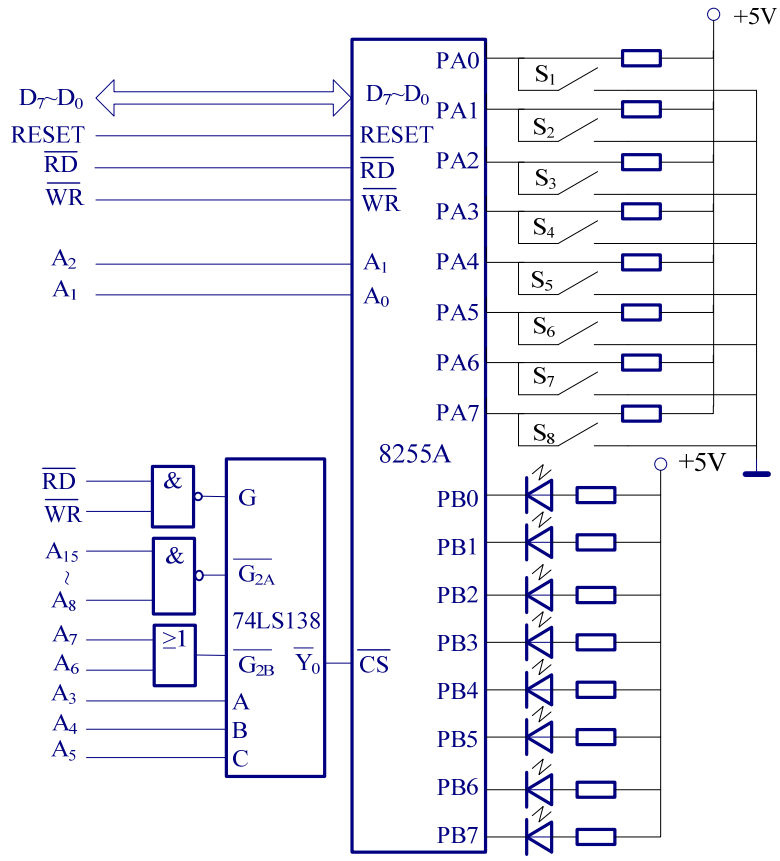


图 7.32　8255A 外接开关和发光二极管

解　8255A 的 A 口和 B 口采用方式 0，其中 A 口为输入，B 口为输出。按图 7.32 的硬件接线，8255A 的 A 口地址为 0FF00H 或 0FF01H，B 口地址为 0FF02H 或 0FF03H，C 口地址为 0FF04H 或 0FF05H，控制端口地址为 0FF06H 或 0FF07H。采用 8086CPU 的程序如下：

```
        MOV DX, 0FF06H
        MOV AL,10010000B
        OUT DX, AL
L:      MOV DX, 0FF00H
        IN AL, DX
        MOV DX,0FF02H
        OUT DX, AL
        JMP L
```

例 7.9　采用 8255A 的数码管显示接口电路如图 7.33 所示，试编程实现采用动态扫描方法在 LED 数码管上循环显示 0000～999。

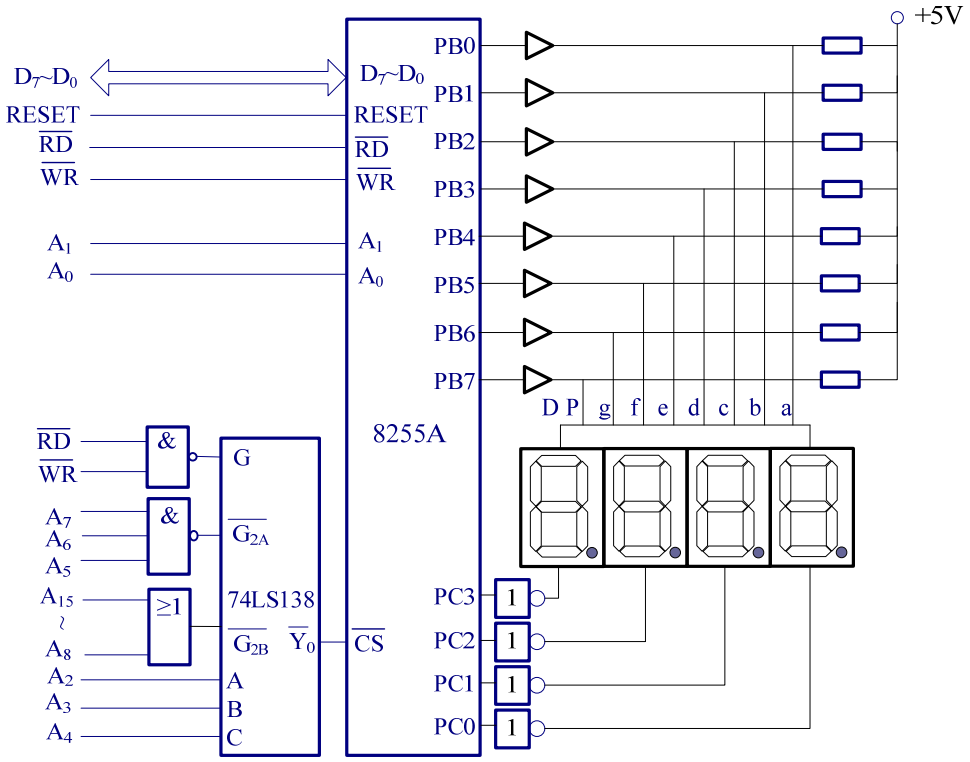


图 7.33　采用 8255A 的数码管显示接口电路

解　LED 数码管的主要部分是发光二极管，通过 7 个发光段的不同组合，可显示 0～9 和 A～F 以及某些特殊字符。由于发光二极管发光时，通过的平均电流为 10～20mA，而通常的输出锁存器不能提供这么大的电流，所以 LED 各段必须接驱动电路，见图 7.33。

点亮数码管有静态和动态两种方法。所谓静态显示，就是当数码管显示某一个字符时，相应的发光二极管恒定地导通或截止。这种显示方式每一个数码管都需要有一个 8 位输出口控制，而当系统中数码管较多时，用静态显示所需的 I/O 口太多，一般采用动态显示方法。所谓动态显示就是一位一位地轮流点亮各位数码管(扫描)，对于每一位数码管来说，每隔一段时间点亮一次。数码管的亮度既与导通电流有关，也与点亮时间和间隔时间的比例有关。调整电流和时间参数，可实现亮度较高较稳定的显示。这种显示方法需有两类控制端口，即位控制端口和段控制端口。位控制端口控制哪个数码管显示，段控制端口决定显示代码。所有数码管共用段控制端口公，当 CPU 输出一个显示代码时，各数码管的输入段都收到此代码。但是，只有位控制码中选中的数码管才得到导通而显示。

8255A 的 A 口、B 口、C 口地址分别为 0E0H、0E1H、0E2H，控制端口地址为 0E3H。A 口、C 口设置为方式 0 输出。

采用 8086CPU 的程序编写如下：

```
DATA    SEGMENT

OUTBUFF    DB 4DUP(?)

LEDTAB DB 3FH, 06H, 5BH, 4FH, 66H, 6DH, 7DH, 07H, 7FH, 6FH

COUNT DB 100
```

```
DATA    ENDS
CODE    SEGMENT
ASSUME    CS: CODE, DS: DATA
START:    MOV       AX, DATA
MOV       DS, AX
MOV       AL, 80H
OUT       0E3H, AL       ;8255A 初始化
MOV       BX,0           ;初始化显示值
NEXT:     LEA       SI, OUTBUFF
MOV       AX, BX         ;将显示值转换为十进制数并保存
MOV       DX, 0
MOV       CX,1000
DIV       CX
MOV       [SI], AL
INC       SI
MOV       AX, DX
MOV       CL,100
DIV       CL
MOV       [SI], AL
INC       SI
MOV       AL, AH
MOV       AH, 0
MOV       CL, 10
DIV       CL
MOV       [SI],AL
INC       SI
MOV       [SI],    AH
AGAIN:    MOV       CH,08H       ;初始化位选码
LEA       SI, OUTBUFF
LEDDISP: MOV       AL, [SI]       ;取显示值
```

```
MOV     AH, 0
LEA     DI, LEDTAB
ADD     DI, AX
MOV     AL, [DI]        ;转换为段码
OUT     0E1H, AL        ;输出段码
MOV     AL,CH
OUT     0E2H, AL        ;输出位选码
CALL    DELAY           ;延时 2 ms
INC     SI
ROR     CH, 1           ;指向下一个数码管
CMP     CH, 80H
JNZ     LEDDISP     ;判该轮是否显示结束
DEC     COUNT       ;重复显示某数 100 次,便于看清该数
JNZ     AGAIN
MOV     COUNT, 100
INC     BX              ;显示数值加 1
CMP     BX, 10000
JZ      EXIT
JMP     NEXT
EXIT:   MOV     AH, 4CH     ;返回 DOS
INT     21H
DELAY   PROC    NEAR        延时子程序
PUSH    BX
PUSH    CX
MOV     BX,10
DELl:   MOV     CX, 0
DEL2:   LOOP    DEL2
DEC     BX
JNZ     DELl
POP     CX
```

```
POP     BX
RET
DELAY   ENDP
CODE    ENDS
END     START
```

在上述程序中，延时子程序中 BX，CX 的初始值随不同型号计算机应作相应调整。

例 7.9　用查询方式工作的打印机接口。8255A 与系统及打印机的连接如图 7.34 所示，并通过该打印机接口打印字符串，字符串长度放在数据段的 COUNT 单元中，要打印的字符存放在从 DATA 单元开始的数据区中。打印机的工作时序如图 7.35 所示。

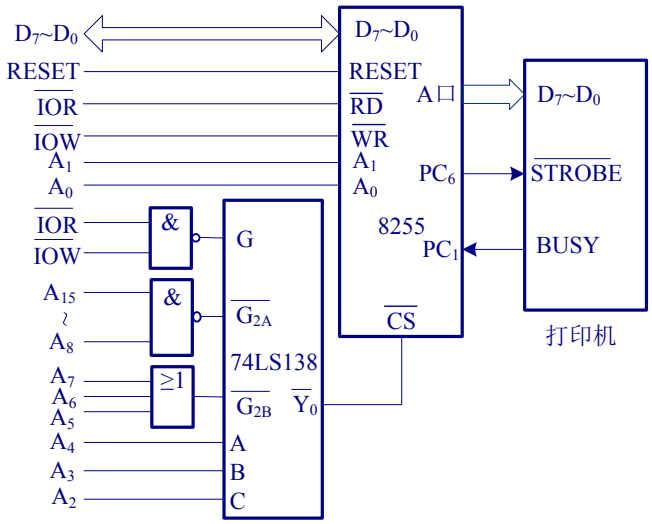


图 7.34　8255A 与打印机的连接

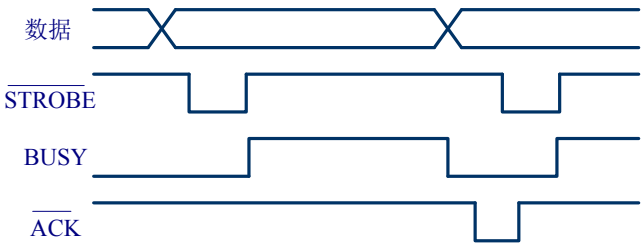


图 7.35　打印机工作时序图

解　由图 7.34 和图 7.35 可知，CPU 通过 8255A 接口将数据传送到打印机的 D_7～D_0 端，然后利用一个负脉冲 $\overline{\text{STROBE}}$ 将数据锁存在打印机内部，以便打印机进行处理。同时，打印机的 BUSY 端送出高电平信号，表示其正忙。仅当 BUSY 端信号变低电平后，CPU 才可以将下一个数据送给打印机。

根据上述需求，本例中对 8255A 的 3 个口均设置为工作方式 0：

① A 口设置为输出口，向打印机输出数据。

② C 口的 PC_6 用做选通输出，与 $\overline{STROBE}$ 连接；PC_1 用做状态输入，与打印机的忙信号(BUSY)连接。因此，可通过初始化使 C 口的高 4 位为输出，C 口的低 4 位为输入。另外，由于数据输出后要通过 PC_6 端输出一个负脉冲，故在初始化时先要将 PC_6 初始化为高电平。

③ B 口不使用，初始化时可任意定义为输入或输出(本例中将 B 口定义为输出口)。

8255A 的地址范围为 0FF00H～0FF03H。由此可得出 8255A 的初始化程序如下：

```
INIT:   MOVDX, FF03H          ;8255 的控制寄存器端口地址送 DX
        MOV AL, 10000001B     ;A 组方式 0：A 口输出，C 口高 4 位输出
                              ;B 组方式 0：B 口输出，C 口低 4 位输入
        OUT DX, AL            ;方式控制字送控制寄存器
        MOV AL, 00001101B     ;C 口的按位操作控制字，使 PC6 初始状态置为 1
        OUT DX,AL     ;C 口位操作控制字送控制寄存器
```

下面是打印一批字符的程序段：

```
        MOV CX, COUNT         ;将字符串长度作为循环次数
        MOV SI, OFFSET DATA   ;取字符串首地址
GOON:   MOV DX，0FF02H        ;0FF02H 为 C 口的地址
        IN AL, DX             ;从 C 口读入打印机的 BUSY 信号状态
        AND AL, 02H           ;测试打印机状态(位 1)
        JNZ GOON              ;若 BUSY 为高电子，则循环等待
        MOV AL, [SI]          ;否则取一个字符
        MOV DX, 0FF00H        ;0FF00H 为 A 口的地址
        OUT DX, AL            ;输出一个字符到 A 口
        MOV DX, 0FF02H        ;准备在 PC6 上生成一个负脉冲
        MOV AL, 0
        OUT DX, AL      ;因仅 PC6 接打印机，故由 C 口输出 00H 将使 PC6 变低
        MOV AL, 40H
        OUT DX, AL      ;再使 PC6 变高，在 PC6 上生成一个 STROBE 负脉冲
        INC SI            ;指向下一个字符
        LOOP  GOON      ;若未结束，则继续
        HLT
```

上述程序中，在 PC_6 引脚上生成的 $\overline{STROBE}$ 负脉冲是通过往 C 口输出数据(先将 PC_6 初始化为 1，输出一个 0，然后再输出一个 1)而形成的。当然，也可以利用 C 口的位控制字对 PC_6 进行置位/复位操作来实现。程序如下：

```
MOV DX, 0FF03H
MOV AL, 00001100B       ;PC6 复位(=0)
OUT DX, AL
MOV AL, 00001101B       ;PC6 置位(=1)
OUT DX, AL
```

7.4. 串口通信和可编程接口芯片 8251A 及其应用

7.4.1. 串行通信的基本概念

随着信息技术的发展，微型计算机在远程通信领域中的应用日益增多，已经成为人们不可缺少的通信工具。同时，微型计算机本身也带有若干外设(如鼠标、打印机、绘图仪、摄像头等)，需要与它们进行数据通信。在计算机数据传送中，有两种基本的数据传送方式：串行通信和并行通信。采用串行通信时，数据通过一条线路传输，因此可以简化通信设备、降低使用通信线路的价格，并且能利用现有的通信系统。因此，人们为微型计算机制定了适合各类外部设备的接口标准，设计了相应的串行、并行通信接口。

一、串行通信与并行通信

并行通信是指利用多根传输线将多位数据同时进行传送。一字节的数据通过 8 条传输线同时发送。由于并行通信方式使用的线路多，一般用在如计算机与打印机等距离短、数据量大的场合。

串行通信是指利用一条传输线将数据一位一位地按顺序分时传输。当传送一字节的数据时，8 位数据通过一条线分 8 个时间段发出，发出顺序一般是由低位到高位。

串行通信的优势是用于通信的线路少，因而在远距离通信时可以降低通信成本。另外，它还可以利用现有的通信信道(如电话线路等)，使数据通信系统遍布千千万万个家庭和办公室。串行通信适合于远距离数据传送，例如，微型机与计算中心之间、微机系统之间或与其他系统之间。串行通信也由于连线方便而常用于速度要求不高的近距离数据传送，例如，同房间的微型机之间、微型机与绘图机之间、微型机与字符显示器之间。PC 系列上都有两个串行异步通信接口，键盘、鼠标器与主机之间也采用串行数据传送方式。

相对于并行通信方式，串行通信速度较慢。现在，高速的串行通信标准如 USB 接口标准已制定，获得了广泛应用。

二、异步串行通信

串行通信系统中为了使收发数据正确，收发两端操作必须相互协调，即收发在时间上应同步。同步方式有两种：异步串行通信 ASYNC(Asynchronous Data Communication)和同步串行通信 SYNC(Synchronous Data CommunicatiOn)。

异步传送是计算机通信中常用的串行通信方式。异步是指发送端和接收端不使用共同的时钟，也不在数据中传送同步信号。在这种方式下，收方与发方之间必须约定数据帧格式和波特率。

1．数据帧格式

图 7.36 为异步传送的数据帧格式。每帧包括：1 个起始位(低电平)、5～8 个数据位、1 个可选的奇偶校验位、1～2 个终止位(高电平)。

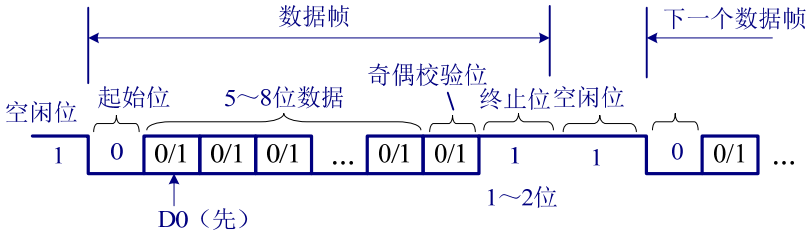


图 7.36　异步通信的数据帧格式

相邻两个数据帧之间的间隔称为空闲位，长度任意，为高电平。由高电平变为低电平就是起始位，后面紧跟的是 5～8 位有效数据位。传送时数据的低位在前、高位在后。数据的后面跟奇偶校验位(可选)，结束是高电平的终止位(1～2 位)。起始位至停止位构成一帧。下一数据帧的开始又以下降沿为标志，即起始位开始。通常 5～8 位数据可表示一个字符，如 ASCII 码就是 7 位。

2．波特率(BaudRate)

波特率是衡量串行数据传送速度的参数，是指单位时间内传送二进制数据的位数，以位/秒为单位(bps，b/s)，也称为波特。PC 中异步串行通信的速度一般为 50 到 19 200 波特之间。常用的波特率有 50、75、100、110、150、300、600、1 200、2 400、4 800、9 600、19 200 等。

由于每一帧开始时将进行起始位的检测，因此收发双方的起始时间是对齐的。收发双方使用相同的波特率，虽然收发双方的时钟不可能完全一样，但由于每一帧的位数最多只有 12 位，因此时钟的微小误差不会影响接受数据的正确性。这就是异步串行通信能实现数据正确传送的基本原理。

例 7.10　设数据帧为 1 位起始位、1 位终止位、7 位数据位、1 位奇偶校验位，传送的波特率为 1 200。用 7 位数据位代表一个字符，求最高字符传送速度。

解　(1 200b/s)/10b=120b/s

例 7.11　设数据帧为 1 位起始位、2 位终止位、8 位数据位、1 位奇偶校验位，要求每秒传送字节数大于 1 kB，则波特率应大于多少波特?

解　(12b/s)×1 000=12 000b/s，波特率应大于 12 000b/s。

三、同步串行通信

异步串行通信中每一帧都需要附加起始位和停止位使数据成帧，因而降低了传送有效数据的效率。对于快速传送大量数据的场合，为了提高数据传输的效率，一般采用同步串行传送。

同步传送时，无需起始位、停止位。每一帧包含较多的数据，在每一帧开始处使用 1~2 个同步字符以表示一帧的开始。一种同步串行通信的数据格式如图 7.37 所示。同步传送要求对传送的每一位在收发两端保持严格同步，发送、接收端可使用同一时钟源以保证同步，或在发送端采用某种编码方式，在收端将时钟恢复。

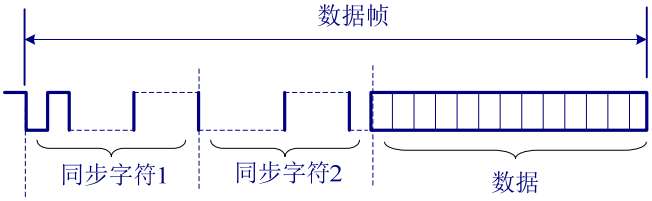


图 7.37　某种同步通信的数据帧格式

四、串行通信中的数据传送模式

串行通信中，数据在两个站 A 或 B 之间传送，有单工、半双工与全双工三种模式。如图 7.38 所示。

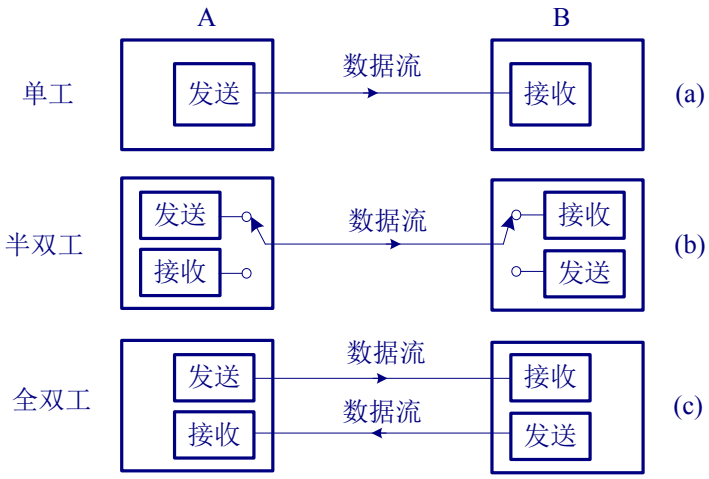


图 7.38　单工、半双工与全双工

1．单工(Simplex)

只能进行一个方向的数据传送。如图 7.38(a)所示，A 作为发送器，B 作为接受器，数据由 A 发送到 B。

2．半双工(Half-Duplex)

这种方式下，A、B 交替地进行双向数据传送。但由于两设备之间只有一条传输线，因此只能分时地进行收和发，不能同时进行双向数据传送。如图 7.38(b)所示

3．全双工(Full-Duplex)

两设备之间有两根传输线，对于每一个设备来讲都有一条专用的发送线和一条专用的接收线，因此可以同时实现双向数据传送。如图 7.38(c)所示。

五、信号的调制和解调

如果直接以逻辑电平表示的数字信号进行传送，由于其频谱很宽，需要的通信线路的频带也就很宽。

在进行远程数据通信时，通信线路往往是借用现有的公用电话网或其他通信网络。而现有的通信网的带宽是一定的，如电话线路的带宽是 3.4 kHz，因此不合适直接传输二进制数据。为了利用电话线传输数字信号，必须采取一些措施，把数字信号转换为适合传输的模拟信号，而在接收端再将其转换成数字信号。前一种转换称为调制，后一种转换称为解调。完成调制、解调功能的设备称为调制解调器(Modem)。应选择合适的调制方式，使调制器输出的信号频谱在通信线路的频带范围内，同时又具有较高的数据传输率，并且接收端易于解调。

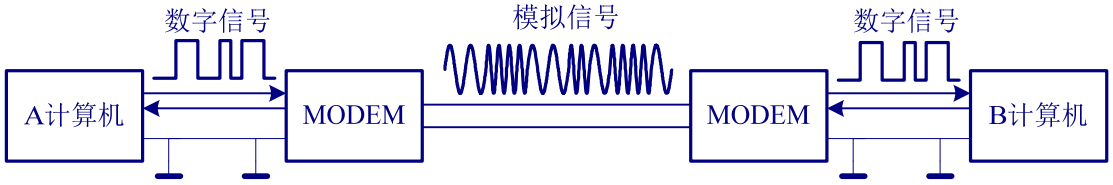


图 7.39　调制和解调

图 7.39 中 Modem 即调制解调器(Modulator-Demodulator)，能实现数字信号的调制和解调。调制的方式有幅移键控 ASK(Amplitude Shift Keying)、频移键控 FSK(Frequency Shift Keying)和相移键控 PSK(Phase Shift Keying)等。图 7.40 所示为一种 ASK 调制方式，在该种方式中，用一种频率的正弦波表示数字 1，用幅值为零的正弦波表示 0。图 7.41 所示为一种 FSK 调制方式，在该种方式中，用一种频率的正弦波表示数字 1，用另一种频率的正弦波表示 0。

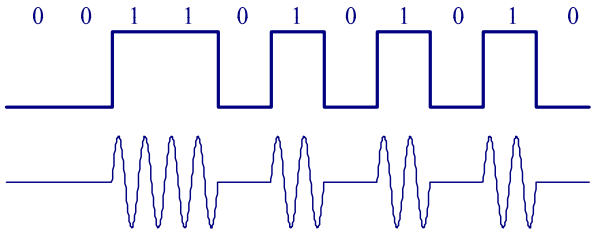


图 7.40　ASK 调制方式

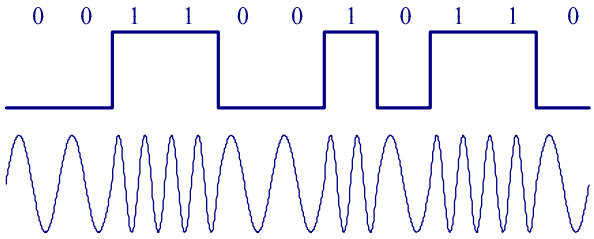


图 7.41　FSK 调制方式

调制解调器大体上可分为两类，一类是异步调制解调，另一类是同步调制解调。异步调制解调适用于异步通信方式，常用的调制方式是频移键控 FSK。其基本原理是将“0”和“1”两种数字信号以不同频率的信号表示，如图 10．5 所示。同步调制解调器主要用于高速同步通信，常用的方式是调相和正交幅度调制等。PSK 的调制方式与 FSK 相比，使用的频带较窄，传送速率高，抗干扰性能强，因此高速调制解调器一般采用 PSK 调制。如 V．92 标准

的调制解调器，可利用电话线上网，其通信速率达到 56 Kbps。

六、串行接口标准日 RS-232C

RS-232C 是得到广泛使用的串行异步通信接口标准。它是美国电子工业协会(Electronic Industry Association，EIA)于 1962 年公布，并于 1969 年修订的串行接口标准。事实上已经成为国际上通用的标准串行接口。1987 年 1 月，RS-232C 经修改后，正式改名为 EIA-232D。由于标准修改并不多，因此，现在很多厂商仍沿用旧的名称。

最初，RS-232C 串行接口的设计目的是用于连接调制解调器。目前，RS-232C 已成为数据终端设备 DTE(例如计算机)与数据通信设备 DCE(例如调制解调器)的标准接口。利用 RS-233C 接口不仅可以实现远距离通信，也可以近距离连接两台微机或电子设备。

1．RS-232C 的引脚定义

RS-232C 接口标准使用标准的 25 针 D 型连接器即 DB-25。表 7.6 罗列了它的引脚排列和名称。PC 已使用 9 针连接器即 DB-9 取代 25 针连接器，因此表 7.6 中也给出了 9 针连接器的引脚。图 7.42 为 25 针连接器和 9 针连接器。

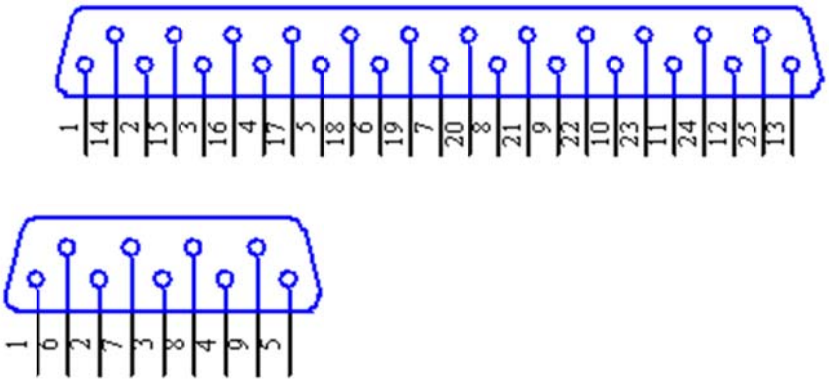


图 7.42　DB-25 连接器和 DB-9 连接器

表 7.6　RS-232C 的引脚定义

9 针连接器引脚	25 针连接器引脚	名 称	25 针连接器引脚	名 称
	1	保护地	12	次信道载波检测
3	2	发送数据 TxD	13	次信道清除发送
2	3	接收数据 RxD	14	次信道发送数据
7	4	请求发送 RTS	16	次信道接收数据
8	5	清除发送 CTS	19	次信道请求发送
6	6	数据装置准备好 DSR	21	信号质量检测

5	7	信号地 GND	23	数据信号速率选择
1	8	载波检测 CD	24	终端发生器时钟
4	20	数据终端准备好 DTR	9、10	保留
9	22	振铃提示 RI	11	未定义
	15	发送时钟 TxC	18	未定义
	17	接收时钟 RxC	25	未定义

RS-232C 接口包括两个信道：主信道和次信道。次信道为辅助串行通道提供数据控制和通道，但其传输速率比主信道要低得多，其他跟主信道相同，通常较少使用。

TxD 发送数据——串行数据的发送端。

RxD 接收数据——串行数据的接收端。

RTS 请求发送——当数据终端设备准备好送出数据时，就发出有效的 RTS 信号，用于通知数据通信设备准备接收数据。

CTS 清除发送——当数据通信设备已准备好接收数据终端设备的传送数据时，发出 CTS 有效信号来响应 RTS 信号，其实质是允许发送。

RTS 和 CTS 是数据终端设备与数据通信设备间一对用于数据发送的联络信号。

DTR 数据终端准备好——通常当数据终端设备一加电，该信号就有效，表明数据终端设备准备就绪。

DSR 数据装置准备好——通常表示数据通信设备(即数据装置)已接通电源连到通信线路上，并处在数据传输方式，而不是处于测试方式或断开状态。

DTR 和 DSR 也可用做数据终端设备与数据通信设备问的联络信号，例如，应答数据接收。

GND 信号地——为所有的信号提供一个公共的参考电平。

CD 载波检测——当本地调制解调器接收到来自对方的载波信号时，就从该引脚向数据终端设备提供有效信号。该引脚缩写为 DCD。

RI 振铃指示——当调制解调器接收到对方的拨号信号期间，该引脚信号作

为电话铃响的指示，保持有效。

保护地(机壳地)——这是一个起屏蔽保护作用的接地端，一般应参照设备的使用规定，连接到设备的外壳或机架上，必要时要连接到大地。

TxC 发送器时钟——控制数据终端发送串行数据的时钟信号。

RxC 接收器时钟——控制数据终端接收串行数据的时钟信号。

2．RS-232C 的连接

图 7.43 是数字终端设备(例如微机)利用 RS-232C 接口连接调制解调器的示意图，用于实现通过电话线路的远距离通信。实际上，数据终端设备与数据通信设备通过 RS-232C 接口就是对应引脚直接相连。图中只使用 9 个常用信号，并给出了 DB-9 连接器的引脚号。

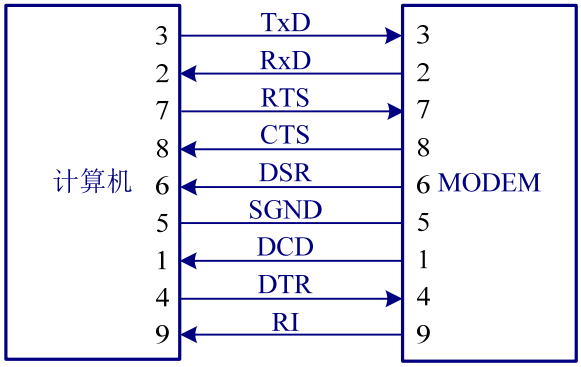
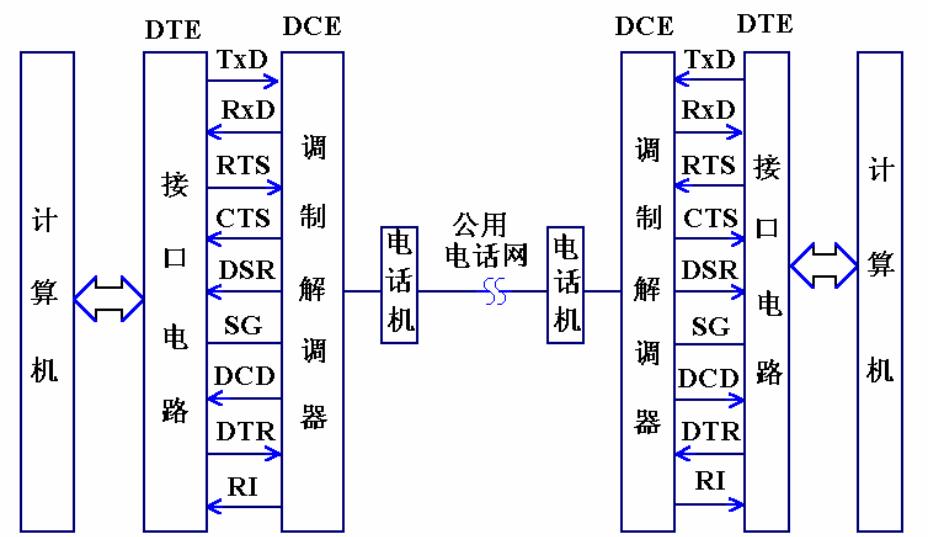
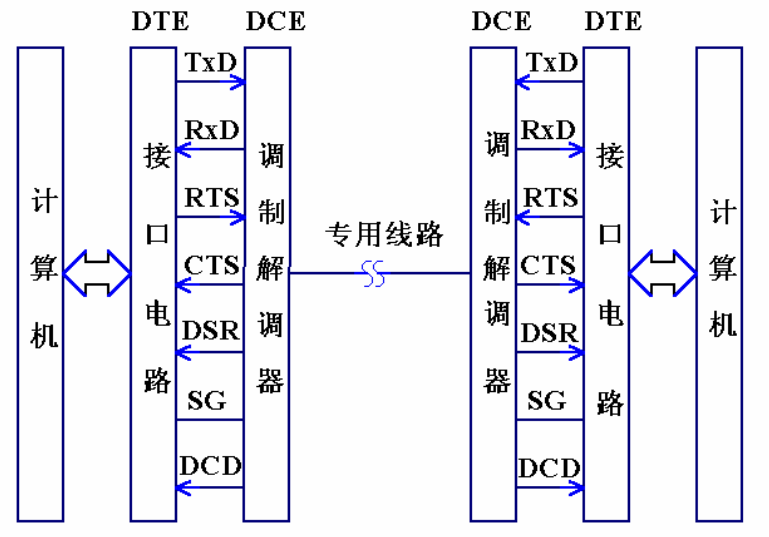


图 7.43　计算机由 RS-232C 接口连接调制解调器

图 7.44 是两台微机直接利用 RS-232C 接口进行长距离通信的连接示意图。被传输的数字信号通过 MODEM 调制为模拟信号，通过公用电话线[图 7.44(a)]或专用线路[图 7.44(b)]进行远距离传输，再经过 MODEM 解调为数字信号，由终端设备（计算机）接受。



(a)



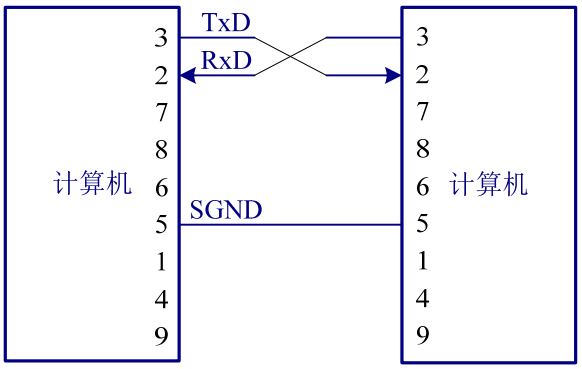
(b)

图 7.44　两台微机直接利用 RS-232C 进行长距离传输

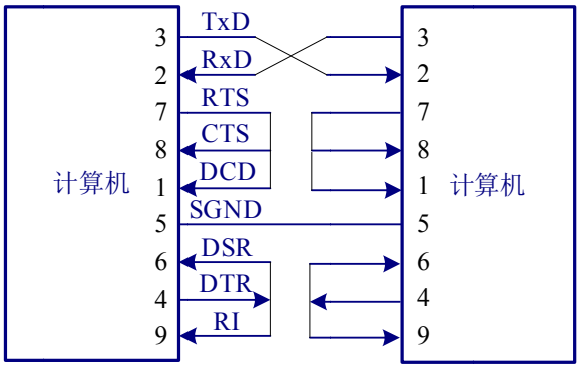
图 7.45 是两台微机直接利用 RS-232C 接口进行短距离通信的连接示意图。由于这种连接不使用调制解调器，所以被称为零调制解调器(Null Modem)连接。

图 7.45(a)是不使用联络信号的 3 线相连方式。很明显，为了交换信息，TxD 和 RxD 应当交叉连接。程序中不必使 RTS 和 DTR 有效，也不应检测 CTS 和 DSR 是否有效。

图 7.45(b)是“伪”使用联络信号的 3 线相连方式，是常用的一种方法。图中双方的 RTS 和 CTS 各自互接，用请求发送 RTS 信号来产生允许发送 CTS，表明请求传送总是允许的。同样，DTR 和 DSR 互接，用数据终端准备好产生数据装置准备好。这样的连接可以满足通信的联络控制要求。



(a)



(b)

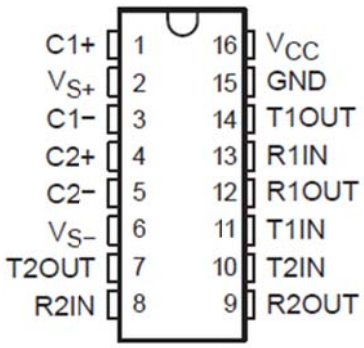
图 7.45　两台微机直接利用 RS-232C 接口进行短距离通信

由于通信双方并未进行联络应答，所以采用图 7.45(a)和图 7.45(b)的连接方式，应注意传输的可靠性。发送方无法知道接收方是否可以接收数据、是否接收到了数据。传输的可靠性需要利用软件来保证，例如程序中先发送一个字符，等待接收方确认之后(回送一个响应字符)再发送下一个字符。

3．RS-232C 的电气特征

RS-232C 接口标准采用 EIA 电平。它规定：高电平为+3~+15V，低电平为-3~-15V。实际应用中常采用±12V 或±15V。RS-232C 可承受最高±25 V 的信号电压。另外，要注意 RS-232C 数据线 TxD 和 RxD 使用负逻辑，即高电子表示逻辑 0，用符号 SPACE(空号)表示；低电平表示逻辑 1，用符号 MARK(传号)表示。联络信号线为正逻辑，高电平有效，为 ON 状态；低电平无效，为 OFF 状态。

由于 RS-232C 的 EIA 电平与微机的逻辑电平(TTL 电平或 CMOS 电平)不兼容，所以两者间需要进行电平转换。传统的转换器件有 MCl488(完成 TTL 电平到 EIA 电子的转换)和 MCl489(完成 EIA 电子到 TTL 电平的转换)等芯片。目前已有更为方便的电平转换芯片，例如 MAX232、UN232 等。MAX232 电平转换电路如图 7.46 所示，其外围元件很少，一块芯片就能实现两路 TTL 电平到 EIA 电平、两路 EIA 电平到 TTL 电平的转换。



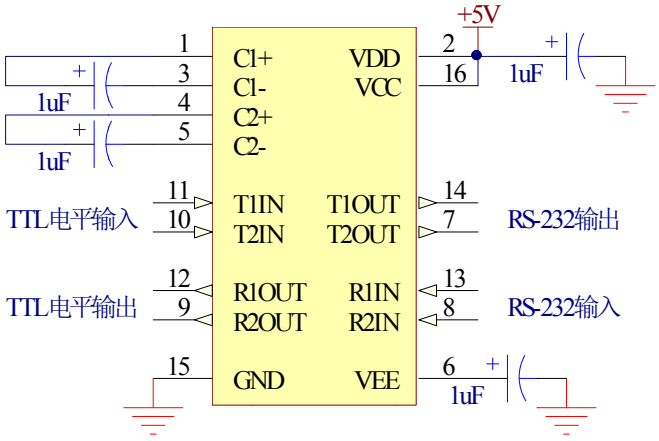


图 7.46　MAX232 的封装和应用电路图

7.4.2. 8251A 的内部结构和外部引脚

Intel 8251A 是 Intel 公司生产的通用可编程串行通信接口芯片，通过编程可实现串行接口的基本任务。其基本功能包括：

1．8251A 是可编程的串行通信接口芯片，能够以同步方式或异步方式进行工作。能自动完成帧格式。

2．在同步方式中，每个字符可定义为 5、6、7 或 8 位，可以选择进行奇校验、偶校验或不校验。内部能自动检测同步字符实现内同步或通过外部电路获得外同步，波特率为 0～64 k。

3．异步方式中，每个字符可定义为 5、6、7 或 8 位，用 1 位作为奇偶校验(可选择)。时钟速率可用软件定义为通信波特率的 1、16 或 64 倍。能自动为每个被输出的数据增加 1 个起始位，并能根据软件编程为每个输出数据增加 1 个、1.5 个或 2 个停止位。异步方式下，波特率为 0～19.2 kbps。

4．8251A 能进行出错检测，它具有奇偶、溢出和帧错误等检测电路。用户可通过输入状态寄存器内容进行查询。

5．具有独立的接收器和发送器，因此，能够以单工、半双工或全双工的方式进行通信。并且提供一些基本控制信号，可以方便地与调制解调器连接。

一、8251A 的内部结构

8251A 的内部结构如图 7.47 所示。8251A 由 5 个主要部分组成，包括接收器、发送器、调制控制、读写控制和系统数据总线缓冲器。8251A 内部由内部数据总线实现相互之间数据传送。

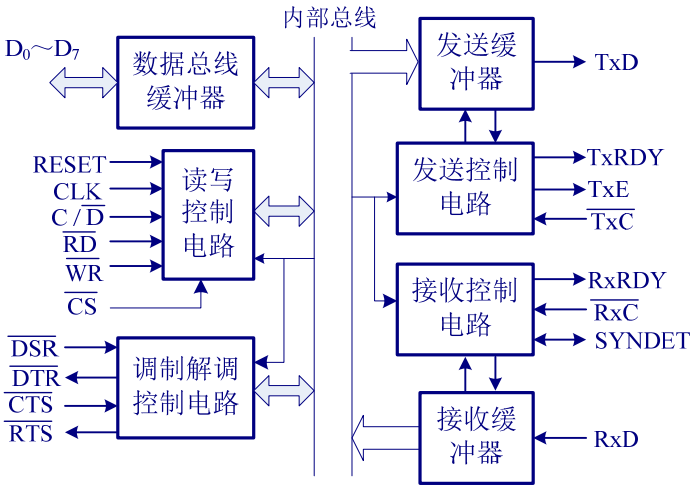


图 7.47　8251A 的内部结构

(1)数据总线缓冲器

数据总线缓冲器是三态双向 8 位缓冲器，它使 8251A 与系统总线连接起来。它含有数据缓冲器和命令缓冲器，CPU 通过输入 / 输出指令对它读/写数据，也可以写入控制字和命令字，再由它产生使 8251A 完成各种功能的控制信号。另外，执行命令所产生的各种状态信息也是从数据总线缓冲器读出的。

(2)接收

接收器的功能是接收在 RxD 引脚上的串行数据，并按规定的格式把它转换　为并行数据，存放到数据总线缓冲器中。

异步方式：且允许接收和准备好接收数据时，它监视 RxD 线。在无字符传送时，RxD 线为高电平。当发现 RxD 线上出现了低电平时，则认为它是一帧信息的起始位，同时启动一个内部计数器，当计数到一个数据位宽度一半时(若时钟脉冲频率为波特率的 16 倍时，则为计数到第 8 个脉冲)，又重新采样 RxD 线，若仍为低电平，则确认它就是起始位，而不是噪声信号。此后，每隔一个数据位宽度时间采样一次。RxD 线作为输入信号，送至移位寄存器，经过移位，又经过去掉停止位和奇偶校验后，变成了并行数据，再经过 8251A 内部数据总线送至接收数据缓冲器，同时发出 RxRDY 信号，通知 CPU 字符已经可以输入了。

鉴于接收器采用上述的方式确定起始位和检测信息，所以在异步串行通信时，大多数可编程串行接口芯片都设计了三种错误类型检测功能。当发送时钟和接收时钟的频率相差太大时，会引起在刚采样几次就造成错位，接收器在收到规定的字符位后，有可能在本应是停止位的数位上出现了低电平，这就是帧格式错；如果接收器对收到的字符位产生的奇偶校验位与收到的奇偶校验位不一致，就是奇偶校验错；当接收器将收到的一个有效字符送至接收数据缓冲器，通知 CPU 读取时，若 CPU 还未读走，又收到一个有效字符时，就会覆盖掉上一个字符，这就是接收缓冲器溢出错误。不管出现哪种类型的错误，接收器均会将所收到的字符数据送至接收数据缓冲器，同时置相应的错误状态标志位。

同步方式：8251A 首先搜索同步字。8251A 检测 RxD 线，每出现一个数位就把它接收下来，并把它送人移位寄存器移位，接收一个整字符后和同步字符寄存器的内容相比较，若不等，还要重复上述部分接收下一个字符继续和同步字符寄存器的内容相比较；若相等，说明 8251A 搜索到了同步字符，此时，8251A 的 SYNDET 引脚就升为高电平以示同步已经实

现。

有时，8251A 采用双同步字符方式。这种情况下，就要测得输入的字符与第一个同步字符寄存器内容相同后，再继续检测此后输入的字符与第二个同步字符寄存器的内容是否相等，如果不等，还要从第一个同步字符寄存器开始比较，若相同，则认为同步已经实现。

在外同步的情况下，和上面过程不同。因为外同步是通过在同步输入引脚 SYNDET 加一个高电位实现同步的。SYNDET 引脚一出现高电平，8251A 就会立即脱离对同步字符的搜索过程，只要此高电平能维持一个接收时钟周期，8251A 便认为已经实现同步了。

8251A 实现同步后，利用时钟采样和移位从 RxD 线上接收来的数据位，且按规定的位数，把它送至接收数据输入缓冲寄存器，同时发出 RxRDY 准备好信号。

(3)发送器

发送器接收 CPU 输出的并行数据，通过移位寄存器，串行从 TxD 引脚输出。

在异步方式时，发送器为每一个字符自动加上 1 个起始位，并且按照编程要求加上奇偶校验位以及 1 个、1.5 个或者 2 个停止位。数据及起始位、校验位、停止位总是在发送时钟 TxC 的下降沿时，从 8251A 发出，数据传输的波特率可以为发送时钟频率的 1、1/16 或者 1/64，具体取决于编程时，选择方式选择字中的波特率系数。

在同步发送方式下，发送器在准备发送的数据前面插入由初始化程序设定的一个或两个同步字符，而在数据中，根据编程设定可插入奇偶校验位。然后，在发送时钟的作用下，以时钟相同的频率将数据一位一位地由 TxD 引脚发送出去。当 8251A 正在发送数据，而 CPU 却来不及提供新数据时，8251A 发送器会自动插入同步字符在 TxD 引脚发出，因为在同步方式时被传送的字符间是不允许存在间隙的。

不论在同步或异步工作方式，只有当程序设置了 TxE 允许发送和 $\overline{\text{CTS}}$ 对调制器发出请求发送的响应信号有效时，才能发送。

(4)调制控制和读写控制

调制控制实现对 MODEM 的控制，读写控制对 CPU 有关控制信号进行译码，用来控制整个 8251A 芯片的工作过程，以完成对数据、状态信息和控制信息的传输。

二、8251A 的芯片引脚

8251A 是用来作为 CPU 与外设或 MODEM 之间的接口的，其引脚如图 7.48 所示。除了地线和电源引脚外，其余的引脚可分为两类，一类是 8251A 和 CPU 之间的信号，另一类是 8251A 和外部设备或调制解调器之间的信号。

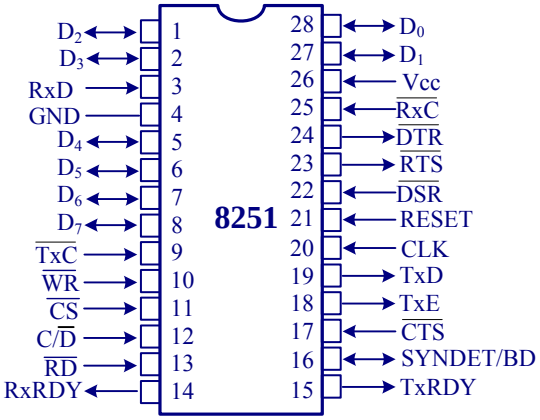


图 7.48　8251A 的芯片引脚

(1)　8251A 与外部装置相连的引脚

1) TxD：发送数据信号端。CPU 送往 8251A 的并行数据，在 8251A 内部转变为串行数据后，通过 TxD 端输出。

2) RxD：接收数据信号端。RxD 用来接收外部装置通过传输线送来的串行数据，数据进入 8251A 后被变换成并行数，等待 CPU 输入。

3) $\overline{\text{DTR}}$：数据终端准备好信号，低电平有效。是由 8251A 送出的一个通用输出信号，它能由初始化 8251A 编程的命令指令的 D_1 置“1”变为有效，用以表示 CPU 准备就绪。

4) $\overline{\text{DSR}}$：数据装置准备好信号，低电平有效。这是一个通用的输入信号，用以表示 MODEM 或外设已经准备好。CPU 可通过读入状态操作，在状态寄存器的 D_7 检测这个信号。一般情况下 $\overline{\text{DTR}}$ 和 $\overline{\text{DSR}}$ 是一组信号，用于接收器。

5) $\overline{\text{RTS}}$：请求发送信号，低电平有效。由 8251A 发送给调制解调器或外设的，CPU 可以通过编程使命令指令的 D_5 置“1”，使其有效，以表示 CPU 已经准备好发送。

6) $\overline{\text{CTS}}$：清除发送信号，低电平有效。这是调制器或外设对 $\overline{\text{RTS}}$ 的响应信号，当其有效时，8251A 才能执行发送操作。

以上 4 个信号对于远距离串行通信时，因为要用到调制解调器，故实际上是连接 8251A 和调制解调器的接口信号。当以上信号用于和计算机外部设备连接，外设不要求有联络信号时，这些信号可以不用，但 $\overline{\text{CTS}}$ 应该接地。因为只有 $\overline{\text{CTS}}$ 有效，才能使 TxRDY 为高电平，只有 TxRDY 为高电平时，CPU 才能往 8251A 发送数据。其他三个信号引脚可悬空不用。

(2)　8251A 与 CPU 相连的引脚

1) D_7～D_0：双向数据线，与系统的数据总线相连。8251A 通过它们与 CPU 进行数据传输，包括 CPU 对 8251A 的编程命令和 8251A 送往 CPU 的状态信息。

2) $\overline{\text{CS}}$：片选信号，低电平有效。它是 8251A 芯片的片选信号，由地址总线经地址译码器输出。只有 $\overline{\text{CS}}$ 信号有效，CPU 才能对 8251A 进行读写。当 $\overline{\text{CS}}$ 为高电平时，8251A 未被选中，8251A 的数据线将处于高阻状态。

3) $\overline{\text{RD}}$：读信号，低电平有效。与系统读控制线相连，当 $\overline{\text{RD}}$ 有效时，CPU 可以从 8251A 的数据口中读取数据或从状态口读取状态信息。

4) $\overline{WR}$：写信号，低电平有效。与系统写控制线相连，当$\overline{WR}$有效时，CPU 可以向 8251A 的控制口写入控制字或向数据口写人数据。

5) C/$\overline{D}$：控制/数据信号。用来区分当前读写的是数据还是控制信息或状态信息，一般与地址总线的最低位 A_0 相连。当 C/$\overline{D}$为高电平时，选中控制端口或状态端口，C/$\overline{D}$为低电平时，选中数据端口。

8251A 共有两个端口地址，数据输入端口和数据输出端口合用一端口地址；状态端口和控制端口合用一个端口地址，它们由$\overline{RD}$和$\overline{WR}$信号区别开。总之，$\overline{CS}$、$\overline{RD}$、$\overline{WR}$和 C/$\overline{D}$这 4 个信号能决定 CPU 对 8251A 的具体操作。

6) TxRDY：发送器准备好信号，高电平有效。它通知 CPU，8251A 的发送器已经准备好，可以接收 CPU 送来的数据，当 8251A 收到一个数据后，TxRDY 信号变为低电平。

当$\overline{CTS}$为低电平，工作命令字中的 TxEN 为高电平，且发送缓冲器为空时，TxRDY 有效。当使用查询方式时，CPU 可以从状态寄存器的 D_0 位检测这个信号；当使用中断方式时，则 TxRDY 可以作为中断请求信号。

7) TxE：发送器空信号，高电平有效。它表示 8251A 发送器已空，即当一个数据发送完成后 TxE 变高。当 CPU 向 8251A 写入一个字符时，TxE 变成低电平。

8) RxRDY：接收器准备好信号，高电平有效。它表示当前 8251A 已经从外部设备或调制解调器上接收到一个字符，正等待 CPU 取走。在中断方式下，该信号可以作为中断请求信号；在查询方式下，该信号可以作为状态信号供 CPU 查询。当 CPU 从 8251A 的数据口读取了一个字符后，RxRDY 变为低电平，表示无数据可取；当 8251A 又收到一个字符后，RxRDY 再次变为高电平。

9) SYNDET/BD：同步和间断检测检测信号。SYNDET/BD 既可以是输入，又可以是输出，它取决于 8251A 是工作在内同步方式，还是工作在外同步方式。

当 8251A 工作在内同步方式时，SYNDET/BD 作为输出端，一旦 8251A 搜索到所要求的同步字符，则 SYNDET/BD 变为高电平，表示 8251A 当前已经达到同步。在双同步字符情况下，SYNDET/BD 会在第二个同步字符的最后位被检测到后，变为高电子。在 CPU 执行一次读操作后，变为低电平。

8251A 工作在外同步方式时，SYNDET/BD 作为输人端，当片外的检测电路找到同步字符后，从这个输入端输入一个正脉冲作为启动脉冲，使 8251A 在 RxC 的下降沿开始拼装字符。SYNDET/BD 的高电平状态最少要维持一个$\overline{RxC}$周期，直到$\overline{RxC}$出现一个下降沿。在外同步方式下，SYNDET/BD 的电平信号取决于外部启动信号。在复位时，SYNDET/BD 变为低电平。

10) RESET：芯片复位线，高电平有效。当 RESET 上有大于等于 6 倍时钟宽度的高电平时，芯片被复位处于空闲状态，直到新的编程命令到来。RESET 通常与系统复位线相连。

(3) 时钟引脚

1) $\overline{RxC}$：接收器时钟输入端。$\overline{RxC}$控制 8251A 接收器接收字符的速度。

在同步方式下，$\overline{RxC}$等于波特率，由调制解调器供给(近距离不用调制解调器传送时由用户自行设置)。在异步方式下，$\overline{RxC}$是波特率的 1、16 或 64 倍，由方式控制命令预先选择。接收器在$\overline{RxC}$的上升沿采集数据。

2) $\overline{\text{TxC}}$：发送器时钟输入端。$\overline{\text{TxC}}$ 控制 8251A 发送器发送字符的速度。时钟频率和波特率之间的关系同 $\overline{\text{RxC}}$，数据在 $\overline{\text{TxC}}$ 的下降沿由发送器移位输出。

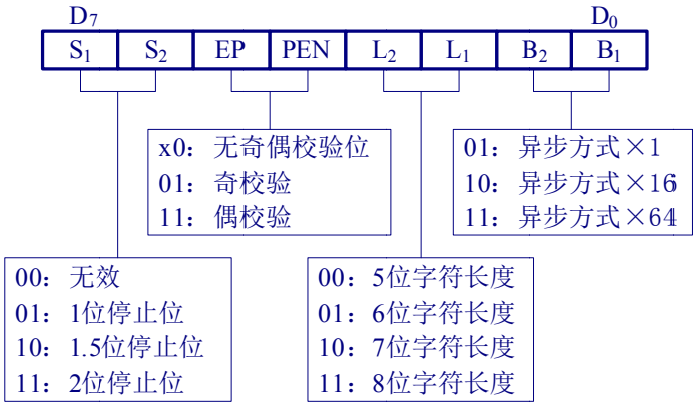
3) CLK：8251A 内部工作时钟信号。由这个 CLK 输入产生 8251A 的内部工作时序。为了使芯片工作可靠，在同步方式工作时 CLK 的频率应大于接收器和发送器时钟频率的 30 倍；在异步方式工作时，CLK 的频率应大于接收器和发送器时钟频率的 4.5 倍。

7.4.3. 8251A 的编程命令

CPU 可以向 8251A 写入控制命令，包括通信方式选择命令和工作命令。CPU 还可以读取 8251A 的状态。下面分别加以说明。

一、通信方式选择命令字

通信方式选择命令字格式如图 7.49 所示，可以可分为 4 组，每组两位。下面分别说明其功能。



(a) 异步方式控制字

(b) 异步方式控制字

图 7.49　8251A 通信方式选择命令字

在通信方式选择命令字中，B_2、B_1 用于确定工作于同步方式还是异步方式；L_2、L_1 用于选择字符的位数；EP、PEN 用于确定奇偶校验方式；D_7、D_6 在异步方式下为 S_1、S_2 位，用于定义停止位的位数，在同步方式下为 SCS、ESD 位，用于定义同步方式。

例如，若要求 8251A 作为异步通信的接口，且波特率系数为 16，7 位字符，采用偶校验，2 个停止位。其方式选择字为：

110111010B=0DAH

若要求 8251A 作为同步通信的接口，内同步且需 2 个同步字符，偶校验，7 位字符。其方式选择字为：

00111000B=38H

二、工作命令字

工作命令字用于确定 8251A 的操作，使 8251A 处于某种工作状态，以便接收或发送数据。工作命令格式见图 7.50。

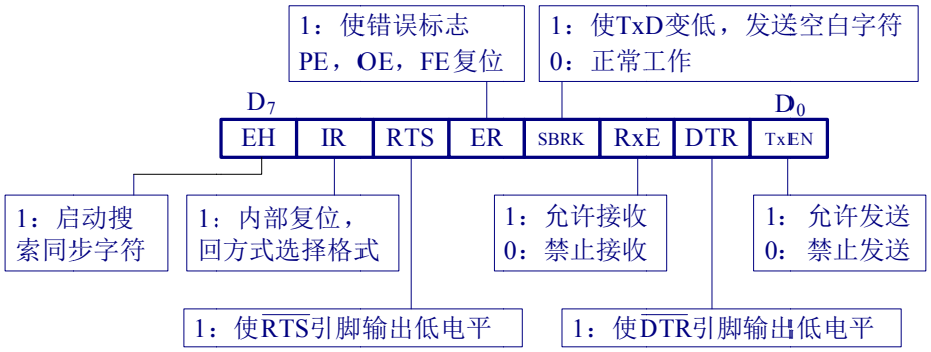


图 7.50　8251A 工作命令格式

三、工作状态字

8251A 进行数据传输后的状态字存放在状态寄存器中，CPU 通过读操作读入状态字，进行分析和判断，以决定下一步的工作。状态字格式如图 7.51 所示。

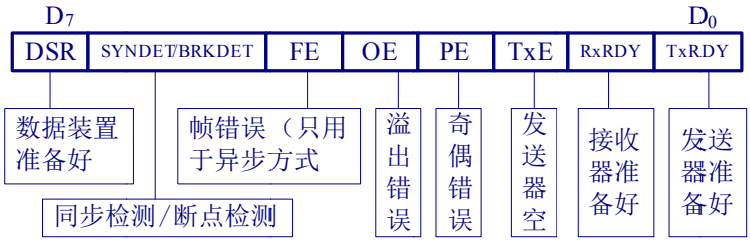


图 7.51　8251A 工作状态字

在工作状态字中，状态位 RxRDY、TxE、SYNDET 的定义与 8251A 芯片引脚的定义完全相同，DSR 与芯片引脚 $\overline{DSR}$ 意义相同，但有效电平相反。状态位 TxRDY，只要发送数据缓冲寄存器一空就置 1，而 8251A 芯片引脚 TxRDY 为高电平的条件，除发送数据缓冲寄存器空外，还必须满足 $\overline{CTS}$ =0 和 TxE=1 两个条件。

另外，PE 为奇偶校验错误标志位，OE 为溢出错误(也称为覆盖错误)标志位，TE 为帧格式错误标志位。所有状态位均置 1 有效。

7.4.4. 8251A 的初始化编程

8251A 是可编程的多功能串行通信接口，在使用 8251A 时，必须先对它进行初始化编程，选择 8251A 的工作方式等。

一、初始化编程的步骤

(1) 芯片复位后，第一次用输出指令写入奇地址端口的应是方式选择指令。约定双方的通信方式(同步／异步)，数据格式(数据位和停止位长度、校验特征、同步字符特征)及传输速率(波特率系数)等参数。

(2) 如果方式选样指令中规定了 8251A 工作在同步方式，那么，CPU 用执行输出指令向奇地址端口写入规定的 1 个或 2 个字节的同步字符。

(3) 只要不是复位命令，不论同步方式还是异步方式，均由 CPU 执行输出指令向奇地址端口写人工作命令指令，控制允许发送／接收或复位。

初始化结束后，CPU 就可通过查询 8251A 的状态字内容或采用中断方式，进行正常的串行通信发送/接收工作。

因为方式字、命令字及同步字均无特征标志位，且都是写入同一个命令口地址，所以在对 8251A 初始化编程时，必须按一定的顺序流程，若改变了这种顺序流程，8251A 就不能识别。

二、内部复位命令

当 8251A 通过写入方式选择字，规定了 8251A 的工作方式后，可以根据对 8251A 工作状态的不同要求随时向控制端口输出工作命令指令字。若要改变 8251A 工作方式，应先使 8251A 芯片复位，内部复位命令字为 40H。8251A 芯片复位后，又可重新向 8251A 输出方式选择字，以改变 8251A 的工作方式。

下面具体介绍初始化 8251A 的方式选择命令字和工作命令字。

1．异步方式下的初始化编程

要求使 8251A 工作在异步方式，波特率系数为 16，字符长度为 8 位，偶校验，2 个停止位。则方式选择字为：11111110B：0FEH。工作状态要求：复位出错标志、使请求发送信号 $\overline{\text{RTS}}$ 有效、使数据终端准备好信号 $\overline{\text{DTR}}$ 有效、发送允许 TxEN 有效、接收允许 RxE 有效。则工作命令指令字应为 37H。假设 8251A 的两个端口地址分别为 0COH 和 0C2H，初始化编程如下：

```
MOV AL, 0FEH
OUT 0C2H, AL      ;设置工作方式
MOV AL, 37H
OUT 0C2H, AL      ;设置工作状态
```

2．同步方式下初始化编程

要求 8251A 工作在同步方式，两个同步字符(内同步)、奇校验、每个字符 8 位，则方式选择字应为 1CH。工作状态要求：使出错标志复位，允许发送和接收、使 CPU 已准备好且请求发送，启动搜索同步字符，则工作命令指令应该是 0B7H。又设第一个同步字符为 OAAH，第二个同步字符为 55H。还使用上例 8251A 芯片，这样要先用内部复位命令 40H，使 8251A 复位后，再写入方式选择控制字。具体程序段如下：

```
MOV       AL,40H
OUT       0C2H,AL      ;复位 8251A
```

```
MOV     AL,1CH
OUT     0C2H,AL     ;设置方式选择字
MOV     AL,0AAH
OUT     OC2H,AL     ;写入第一个同步字符
MOV     AL,55H
OUT     0C2H,AL     ;写入第二个同步字符
MOV     AL,0B7H
OUT     0C2H,AL     ;设置命令字
```

7.4.5. 8251A 的应用实例

例 7.10　用异步串行输入方式输入 2 000 个数据，存放到内存 BUFFER 开始的单元中。因为 8251A 从外设接收一个字符 RxRDY 会自动置位，因此程序中不断对状态寄存器的 RxRDY 位进行测试，查询 8251A 是否已经从外设接收了一个字符。若 RxRDY 变为有效，即收到一个字符，CPU 就执行输入指令取回一个数据存放到内存缓冲区，RxRDY 在 CPU 输入一个字符后会自动复位。除了对状态寄存器的 RxRDY 位检测之外，程序还要检测状态寄存器的 D_3、D_4、D_5 位，以判断是否出现奇/偶错、覆盖错或帧格式错误，若发现错误就转错误处理程序。下面的程序中没有给出错处程序。设 8251 的地址为 80H 和 81H。

```
MOV AL, 0FEH      ;异步方式选择字
OUT 81H, AL       ;写入
MOV AL, 37H       ;工作命令字
OUT 81H, AL       ;写人
MOV BX,BUFFER   ;BX 指向缓冲区首址
MOV DI, 0         ;变址寄存器初值为 0
MOV CX, 2000      ;设置计数器初值
WAIT:   IN AL, 81H          ;读状态字到 AL
TEST AL, 2        ;测试状态字的J D2 位即 RxRDY 位
JZ WAIT           ;为 0，未收到字符，继续取状态字
IN AL,80H         ;当 RxRDY 为 1，则从数据口输人数据
MOV [BX][D1], AL  ;将字符送入缓冲区
INC DI            ;缓冲区指针下移一个单元
IN AL, 81H        ;读状态字
TEST AL, 38H      ;判断有无三种错误
```

JNZ ERR　　;有错，则转出错处理程序

LOOP WAIT　　;没错，判是否结束循环

JMP EXIT　　;结束

ERR:　CALL ERR_PRO　;转入错误处理程序

EXIT:　　...

例 7.11　图 7.52 是以 8251A 作为异步串行接口的电路图。8251A 的发送时钟 TxC 和接收时钟 RxC 由 8253 的 OUT2 提供。要求 8251A 的波特率为 2 400bps，波特率系数选 16，8251A 的 TxC 和 RxC 应为 38.4 kHz。8251A 的时钟 CLK 频率为 2 MHz。8251A 的 C/$\overline{D}$ 接 A_0，片选信号 $\overline{CS}$ 由 CPU 的地址线 A_{15}～A_1 译码输出，数据端口地址为 0E0H，控制端口地址为 0ElH。由 MAX232 实现电子转换。试完成 8251A 的初始化编程和数据输出编程。

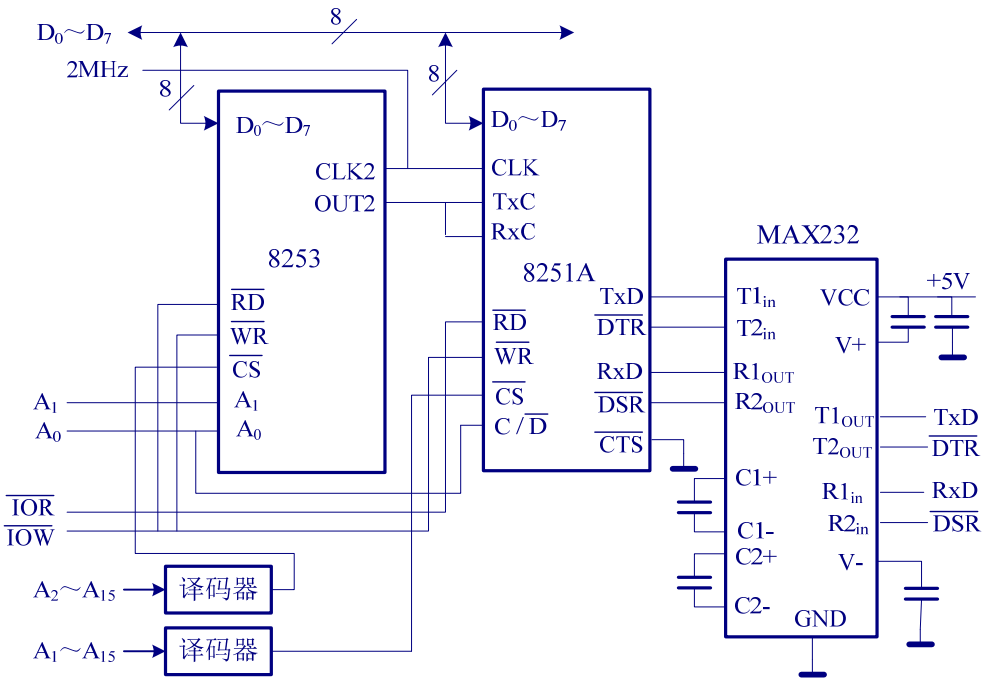


图 7.52　8251A 的应用电路

解　8253 的 CLK 为 2 MHz，使 8253 的计数器 2 工作于方波方式，分频系数为 52，则 OUT2 输出频率约为 38.461 kHz，基本满足要求。

下面给出完成对 8251A 的初始化编程的程序。在对 8251A 设置方式字之前，先进行 8251A 的软件复位，程序中先送三个 0，再送 40H 到控制端口使 8251A 复位。初始化程序段如下：

MOV AL, 00H

OUT 0ElH, AL

CALL DELAY　　;调用延时程序

OUT 0ElH, AL

```
CALL    DELAY
OUT     0ElH, AL
CALL DELAY
MOV AL, 40H       ;内部复位命令指令字
OUT 0ElH, AL      ;确保 8251A 复位
CALL DELAY
MOV AL, 4EH       ;写入方式控制字,异步、8 位数据
OUT 0ElH, AL      ;波特率系数为 16,不校验、1 个停止位
MOV AL, 27H       ;写人命令字,启动发送器和接收器
OUT 0ElH, AL
```

若串行通信采用查询方式，则程序应先对状态字进行测试，判断 TxRDY 状态位是否有效。若 TxRDY 为高电平，说明当前数据输出缓冲器为空，CPU 可以向 8251A 输出该数据；否则就等待。

设要向外输出的数据已放在 AH 寄存器中。数据输出的程序段如下：

```
WAIT:   IN AL, 0ElH         ;读状态口,输入状态字
        TEST AL, 01H        ;测试状态字 D0 位 TxRDY
        JZ WAIT             ;若为 0 则等待
        MOV AL, AH          ;取要发送的数据
        OUT 0E0H, AL        ;往端口输出一个字符
```

习题

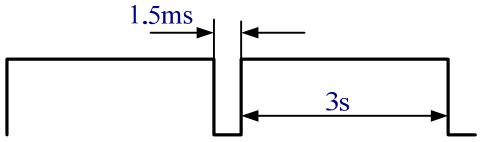
7.1　8253 可编程计数器有两种启动方式，在软件启动时，要使计数正常进行，GATE 端必须为什么电平？如果是硬件启动呢？

7.2　若 8253 芯片的接口地址为 0D0D0H—0D0D3H，时钟信号频率为 2MHz。现利用计数器 0、1、2 分别产生周期为 10μs 的对称方波及每 1 ms 和 1s 产生一个负脉冲，试画出其与系统的电路连接图，并编写初始化程序。

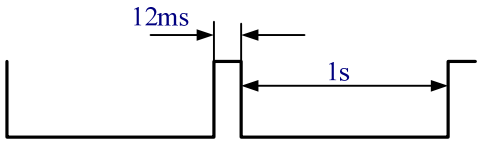
7.3　某一计算机应用系统采用 8253 的计数器 0 做频率发生器，用计数器 1 产生 1 000Hz 的连续方波信号，输入 8253 的时钟频率为 1.2 MHz。试问：初始化时送到计数器 0 和计数器 1 的计数初值分别为多少？计数器 1 工作于什么方式下？

7.4　用 8253 的计数器 2 产生连续脉冲信号，高电平时间为 100μs，低电平时间为 1μs。编写初始化程序，并说明计数器 2 的输入时钟频率是多少。

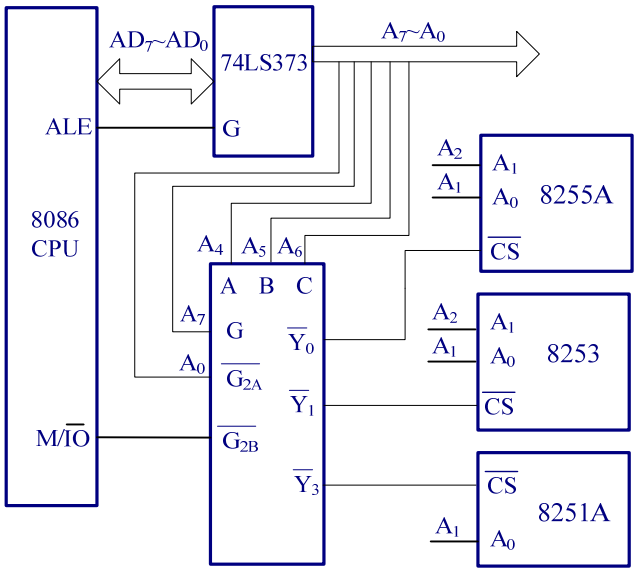
7.5　在以 8086 构成的最大方式系统中，有一片 8254 的端口地址分别为 301H、303H、305H 和 307H，给定的外部时钟为 512kHz。要求：(1) 利用计数器 0 产生周期为 1ms 的周期信号，请编写初始化程序；利用这一计数器能产生的最低信号频率为多少？这时的计时初始值为多少？(2) 利用计数器 1 和 2 产生如下图所示的周期信号，并编写初始化程序。



7.6　设 8253 的端口地址为 260H～263H，外部时钟信号为 1MHz，要求产生如下图所示的周期波形，画出 8253 的连接图，并编写初始化程序段。



7.7　8086 系统接口连接关系如下图所示，试分别确定 8255A、8253 及 8251A 的端口地址。

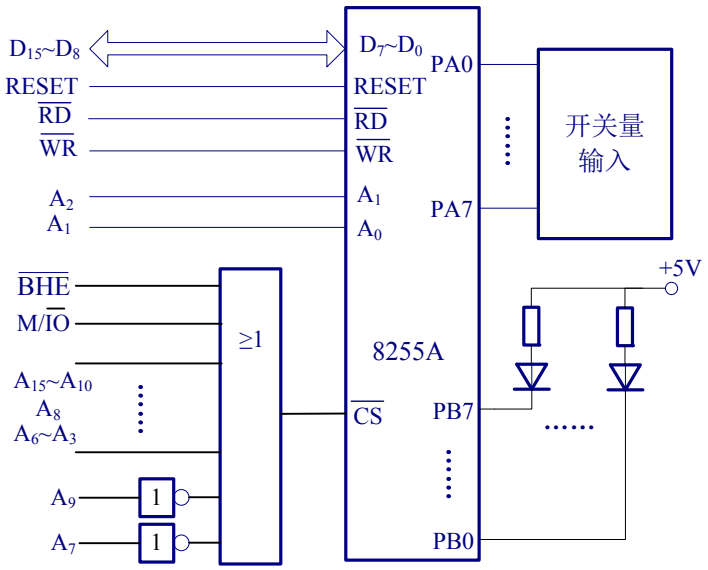


7.8　8255 各端口可以工作在哪几种方式下?当端口 A 工作在方式 2 时，端口 B 和 C 工作于什么方式下？

7.9　某 8255 芯片的地址范围为 0A380H-0A383H，工作于方式 0，A 口、B 口为输出口，现欲将 PC_4 置 0，PC_7 置 1，试编写初始化程序。

7.10 在 8086 最小方式系统中，利用 8255A 某端口输入 8 位开关量，并通过另一个端送出，以发光二极管指示数据，灯亮表示数据“1”，灯灭表示数据“0”。8255A 的端口地址为 280H～287H 中的奇地址，设计系统总线与 8255A 的连接电路，并编程实现。

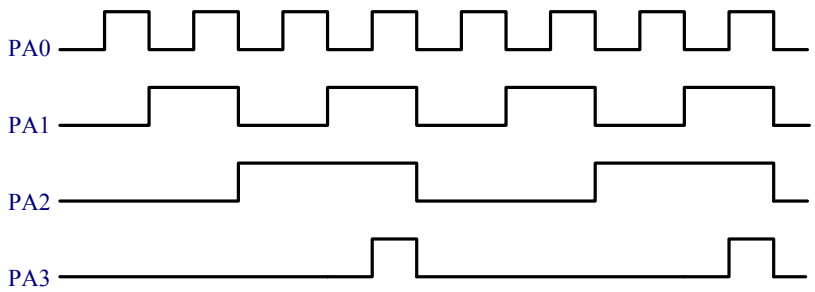
解　按照题目要求，可以采用端口 A 输入开关量（数字量），采用端口 B 输出数据，而且没有增加联络信号的必要，因此可以采用最简单的方式 0。根据 § 10.4 节内容，很容易设计出 8255A 与 8086 最小方式系统的连接关系，如图 10.15 所示，为了使发光二极管具有足够的亮度，我们采用图示的方法连接，这时，当端口 B 的某一位为 0 时，相应的发光二极管亮，这一点可以通过程序进行控制。



8255A 的应用程序段如下：

```
        MOV     DX, 287H                ;设置 8255A 的工作方式
        MOV     AL, 10010000B           ;端口 A 方式 0 输入
        OUT     DX, AL                  ;端口 B 方式 0 输出
RER1:   MOV     DX, 281H                ;从端口 A 读取开关量
        IN      AL,   DX
        NOT     AL                      ;按位取反
        MOV     DX, 283H                ;从端口 B 送出
        OUT     DX,   AL
        JMP     RER1
```

7.11 在 8086 系统中，有一片 8255A，其端口地址为 20H、22H、24H、26H，采用低 8 位地址总线设计译码电路及与系统总线的连接图，并编程实现使端口 A 的低 4 位产生如图所示的信号(各个信号的节拍不必严格相等)。



7.12 什么是同步通信方式？什么是异步通信方式？试说明各自的主要优、缺点，并说明各适合在什么场合下使用。

7.13 在数据通信系统中，什么情况下需采用全双工方式，什么情况下可用半双工方式?

7.14 设异步传输时，每个字符对应 1 个起始位、7 个信息位、1 个奇/偶校验位和 1 个停止位，如果波特率为 9 600bps，每秒能传输的最大字符数为多少个?

7.15 利用一个异步传输系统传送文字资料，系统的波特率为 1 200，待传送的资料为 5 000 个汉字长，设系统不用校验位，停止位只用一位，至少需要多少时间才能传完全部资料?

7.16 在远距离数据传输时，为什么要使用调制解调器？

7.17 全双工和半双工通信的区别是什么？在二线制电路上能否进行全双工通信？为什么？

7.18 在异步传输时，如果发送方的波特率是 600，接收方的波特率是 1200，能否进行正常通信？为什么？

7.19 在 RS-232C 标准中，信号电子与 TTL 电平不兼容，问 RS-232C 标准的 1 和 0 分别对应什么电平? RS-232C 的电平和 TTL 电平之间通常用什么器件进行转换?

7.20 要求 PC 机的 COM2 工作于波特率为 1 200 时，每个字符包含 1 个起始位、7 个信

息位、1 个奇/偶校验位和 1 个停止位，采用偶校验。试完成初始化。

7.21 要求 8251A 工作于异步方式，波特率系数为 16，字符长度为 7 位，奇校验，2 个停 止位。工作状态要求：复位出错标志、使请求发送信号 $\overline{\text{RTS}}$ 有效、使数据终端准备好信号 $\overline{\text{DTR}}$ 有效、发送允许 TxEN 有效、接收允许 RxE 有效。设 8251A 的两个端口地址分别为 0C0H 和 0C2H，试完成始化编程。

参考文献

[1] 徐晨，陈继红，王春明等. 微机原理及应用. 高等教育出版社. 2004

[2] 葛纫秋，韩宇龙，武梦龙. 微型计算机结构与编程. 高等教育出版社. 2005

[3] 冯博琴主编. 吴宁，陈文革，张建编著. 微型计算机硬件技术基础. 高等教育出版社. 2003

[4] 周荷琴 吴秀清. 微型计算机原理与接口技术. 中国科学技术大学出版社. 2008