杜毅仁　李慕靖
王建中　陈启帆　编

16位微型计算机

LIU WEI WEI XING JI SUAN JI

——中册

上海交通大学出版社

十六位微型计算机

（中 册）

杜毅仁 李慕靖 王建中 陈启帆编

上海交通大学出版社

内 容 简 介

本书从微型计算机的硬件、软件、系统结构等方面出发，通过Intel公司的8086系列以及IBM公司的个人计算机作为典型，进行系统的介绍和剖析，使读者获得有关微机的基础和应用知识。

全书共分上、中、下三册。上册通俗易懂地介绍了8086的硬件结构、指令系统及其程序设计语言，是全书的基础和核心部分；中册就整个8086系列，分别介绍了输入/输出处理器8089、专用数值处理器8087、操作系统固件80130、8086的改进型80186和80286，同时，还对MULTIBUS这一标准的系统总线以及8086的其它配套系列芯片作了介绍；下册比较系统地分析了IBM公司的个人计算机，内容包括PC的硬件结构、系统固件及操作系统等。

本书具有硬件、软件相结合的特点，而且既能了解8086，又能由此了解十六位微型计算机的一般原理及应用。因此，既可作为高等院校的教学参考书，也可作为研究院所、厂矿有关研究人员、工程技术人员乃至一般用户的工作手册。

十六位微型计算机
（中册）
杜毅仁 李慕靖 王建中 陈启帆 编
●
上海交通大学出版社出版
上海淮海中路1984弄19号
新华书店上海发行所发行
常熟文化印刷厂排版
交通大学印刷厂印刷

开本：787×1092 毫米 1/16 印张：31.75 字数：758,000
1984年10月第一版 1986年1月第二次印刷
印数：10,000—20,000

定价：5.80元

中册目录

第二篇 iAPX 86/88 微处理器系列

第 二 篇

iAPX86/88 微处理器系列

第八章 iAPX 86/88 系列的形成和发展

§8.1 概 述

随着社会对信息处理能力需求的不断增长，16位微型机的研制和开发，成了微型计算机应用的主流。这种趋势无论在国外还是国内，都已经变得越来越明显。由于16位微型机同8位微型机相比，具有更大的寻址空间和更强的运算、处理能力，所以在复杂的控制和诊断、字处理、通讯终端、图象终端以及在原先由小型机占据的那些领域中，16位微型机已经并且会继续开辟出更为广阔的天地。

自1978年以来，国际市场上各种16位微处理器纷纷涌现，但在所有的16位微处理器中，最早推出微处理器的Intel公司的产品仍然以其独特的风格占据着领先地位。特别是IBM公司宣布在该公司生产的个人计算机（PC）上采用Intel的产品以后，Intel的16位微处理器日益得到各方面的重视。Intel公司从构成系统的高度出发，提供了一系列向上兼容的16位CPU，并且还提供了种类繁多的支持芯片和各种提升系统功能的协处理机，从而在8086的基础上形成了16位微机的iAPX 86/88系列。由于Intel公司是最早推出16位微处理器的厂家，它拥有大量可以直接提供给用户的成熟的软件。例如80130集成的操作系统固件芯片等。

为了把握住16位微型计算机的发展趋势，对Intel公司的iAPX86/88系列的形成和发展作一番研究，对有志于研究、开发和应用16位微型计算机的人来说是会有所裨益的。本章将对iAPX86/88系列的形成和发展作一些探讨，以便使读者对这个完整的系列有一个全面的认识，在这个基础上再分章地进行详细介绍。

§8.2 iAPX 86/88 系列的发展

iAPX86/88系列的基础是8086（即iAPX86或iAPX86/10），它是一种16位的微处理机CPU。8086的主机设计对于8位机有了很大的突破，它的性能较8位机提高了大约10倍，达到了一般小型计算机的水平。由于8086是Intel 16位微型机的基础，我们在上册中对8086的结构、指令系统及程序编制作了极为详尽的论述。从中我们知道8086用4个段寄存器(CS, DS, SS, ES) 作为基础，能直接寻址1兆字节。我们还知道8086的指令系统设计是很有特色的，在8位字长的处理方面，比其他的16位机更为灵活和方便。在寻址方面，由于采用了mod，r/m方式，在最简单的情况下，仅使用一个字节中的5位就能产生出20位物理地址。

8086 的指令短小精悍，能产生简短高效的目的代码，这些都是 8086 的成功之处。但是，由于 8086 出现得最早，因而不可避免地在某些方面存在着缺陷。这主要表现在寻址空间、双字长运算及存贮器保护等方面。针对这些存在的问题以及为了适应日新月异的发展需要，Intel 公司在 8086 的基础上，对 8086 进行了横向和纵向的性能提升，同时利用了 16 位微型机的优势，对 8 位机的性能也进行了大幅度的提升，从而形成了一个完整的 16 位微型机系列。

在 iAPX 86/88 系列中，把单纯的 8086 命名为 iAPX 86/10，当与 8087 配接后称为 iAPX 86/20，当再与 8089 配接后就称为 iAPX86/21，当 8086 与 80130 配接后就称为 iAPX 86/30。这种命名法对于以 8088，80186，80286 作 CPU 的系统也适用。表 8.1 列出了各种处理器的构成。

表 8.1 iAPX 系列的命名

命名 \ 器件组合	8086	80186	80286	8087	80287	8089	80130
iAPX 86/10	1						
iAPX 86/11	1					1	
iAPX 86/20	1			1			
iAPX 86/21	1			1		1	
iAPX 86/30	1						1
iAPX 186/10		1					
iAPX 186/20		1		1			
iAPX 186/30		1					1
iAPX 286/10			1				
iAPX 286/20			1		1		

命名 \ 器件组合	8088	80188	8087	80287	8089	80130
iAPX 88/10	1					
iAPX 88/11	1				1	
iAPX 88/20	1		1			
iAPX 88/21	1		1		1	
iAPX 88/30	1					1
iAPX 188/10		1				
iAPX 188/20		1	1			
iAPX 188/30		1				1

§8.2.1 iAPX 86/88 系列对 8 位机的性能提升

早期的微型计算机系统都是 8 位的，iAPX 86/88 系列采取了向上兼容的策略，使原来为 8 位机编制的软件仍能为新一代的 16 位机所使用。另一方面，由于在许多应用中大量存在着以字节为单位进行的信息交换，这类应用需要大的存贮容量和较高的运算速度，但在外部却只要 8 位数据总线就够了。Intel 的 8088 就是专门设计来满足这种要求的。iAPX 88 (8088) 具有与 iAPX 86 相同的寄存器结构，相同的寻址空间，相同的指令系统，因此它内部就是一个完整的 16 位微处理器。iAPX 88 对外的数据总线是 8 位，从而能与 8 位的微型机系统匹配。显然，由于 iAPX 88 内部是 16 位的 CPU，所以把它作为 8 位的微处理器来应用时，自然会比其他的 8 位机具有更强的指令功能，更大的寻址空间，并且具有更快的处理速度。由此可见，iAPX 88 在 8 位机领域中有着强大的生命力。采用 8088 作 CPU，并不会影响整个系统的造价，但由此而带来的性能提升却是巨大的。这一点在字符处理和事务处理等应用中特别明显。关于 8088 的详细情况可参见第十二章。

§8.2.2 iAPX86/88系列性能的横向提升

iAPX 86/88 系列，作为 16 位微型计算机的 CPU 对 8 位微处理器作为 CPU 构成的微型计算机来讲，性能有了很大的提高。但是，若 CPU 既要完成各种控制、管理，又要执行各种数值计算，还要处理输入/输出，这就难以使整个系统的速度的提高和性能的增长有所突破。Intel 对此采取的策略并不是马上着手去制造速度更高、功能更强的新的微处理器，而是针对着某几方面存在的缺陷，设计了一系列可供用户选择的、在某一方面功能特别强的协处理器。加上这些协处理器以后，8086 就能和这些协处理器并行操作，从而充分发挥了 8086 的潜力。8087，8089，80130 等是使 iAPX 86/88 系列的性能在横向得以提升的主要协处理器。

一、数值数据协处理器 8087 (NPX)

8087 是一种专门用于处理数值数据运算的协处理器。8087 内部含有 8 个 80 位字长的数据寄存器，4 个专用寄存器，它们分别是控制寄存器(16 位)，状态寄存器 (16 位)，指令指示器 (32 位)和数据指示器 (32 位)。8087 具有 62 条指令，分为传送、比较、算术、取常数，超越函数及控制共 6 种类型。表 8.2 列出了其中有关数值运算的指令。

表 8.2 8087 的有关数值运算指令

算术运算	+、−、*、/、$\sqrt{\ }$、比例、求余、舍入、抽选、求绝对值、改变符号
超越函数	tg、 arctg、 2^N-1、 $M\cdot\log_2 N$ $M\cdot\log_2(N+1)$
取常数	0.0、 1.0、 π $\log_2 10$、 $\log_2 e$、 $\log_{10} 2$、 $\log_{10} e$

8087 的指令结构是十分独特的，它借用了 iAPX 86/88 的 ESC 指令中的六个用户可编程序位来构成 (见图 8.1)。因此，8087 的数值运算指令能与原 iAPX 86/88 的指令混合编制在一个完整的程序中，并由 iAPX86/88 的 CPU 来控制整个程序，但由 CPU 和 8087 分别执行各自的指令操作。

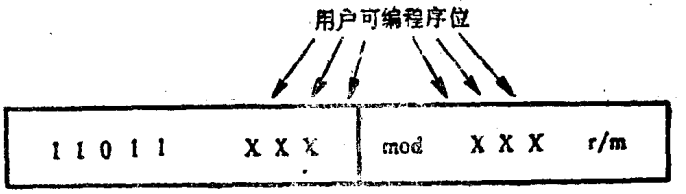


图 8.1 ESC 指令的格式

8087 与 iAPX 86/88 系统的配接是十分方便的。在使用 8087 时，iAPX 86/88 需采用最大模式工作，两种芯片的请求/允许(RQ/GT)端连在一起，并将 8086/88 的队列状态信号 QS_0，QS_1 接入 8087，将 8087 的 BUSY 信号引入 8086 的 TEST 端。运行时，8087 通过 QS_0、QS_1 两个引脚来监视 CPU 中指令队列的状态，并由此对 ESC 指令进行检测。一旦 8087 在数据总线上发现 ESC 指令后，它也同 8086/88 一样将 ESC 指令读入。由于 ESC 中包含了 8087 的数值运算指令，这一过程也就完成了 CPU 向 8087 的指令传递。

8087 的指令都是双存贮器操作数指令，其中一个在系统堆栈的栈顶(或栈顶下第 n 个

数据),另一个由 ESC 中的 "mod,r/m" 指出。当 CPU 根据 "mod,r/m" 信息发出地址并从该单元中读取数据时,8087 也同时将地址或数据读入,从而完成了指令中操作数/操作数地址的传递。

然后,8087 进入数据运算过程,并将 BUSY 置 1。CPU 通过 $\overline{TEST}$ 对 8087 的 BUSY 状态进行监视(使用 WAIT 指令),当发现 8087 运算完毕以后,CPU 再去存贮器中取回结果。

表 8.3 列出了 iAPX86/20(8086+8087) 和 iAPX86,(8086) 处理各种数值运算所需的时间。不难看出,当 8086 加上 8087 组成 iAPX86/20 以后,其系统的数学运算能力将提高 100 倍左右。iAPX 86/20 不但弥补了原 8086CPU 中所缺少的双倍字长以上的各种算术运算功能,而且增加了 32 位、64 位、80 位浮点运算和 18 位 BCD 数据运算的指令。其运算速度之快,运算种类之多都是其他 16 位微处理器所望尘莫及的。关于 8087 的详细情况可参见第十章。

表 8.3 8087 对 CPU 运算速度的提升

运算时间(单位 μs) / 处理机 / 运算类型		iAPX 86/20 (5 MHz) (8086+8087)	iAPX 86/10 (5 MHz) (8086)
浮点运算指令	加/减	14/18	1,600
	单精度乘	19	1,600
	双精度乘	27	2,100
	除	39	3,200
	比较	9	1,300
	取双精度数	10	1,700
	存双精度数	21	1,200
	求平方根	36	19,600
	求正切	90	13,000
	指数运算	100	17,100
18位BCD运算指令	加/减	127	12,040
	乘	297	22,990
	除	323	26,560
	比较	150	20,250

二、输入/输出协处理器 8089 (IOP)

8089 是一种专门用于处理输入/输出的协处理器。8089 共有 52 条指令,1 兆字节的寻址能力,两个独立的 DMA 通道。当 8089 与 iAPX 86/88 的 CPU 配接以后,它也能象 CPU 一样地执行程序,从而代替 CPU 来负责对各种 I/O 外设进行管理。显然,利用 8089 处理 I/O 的方式,已使微型计算机从最早的 TTL 随机逻辑控制和目前广泛使用的单片接口控制片(如 8251,8255 等)提高到了通道控制方式,而这种方式目前只有大中型计算机才普遍采用。8089 使微

处理机的 I/O 处理进入了第三代。与前二种方式相比，随机逻辑接口 I/O 工作时，CPU 需要对外设的每一个 I/O 电平进行控制；单片接口控制片虽然有所改进，但仍然在相当程度上需要依靠 CPU 的支持，因而几乎每读入/发送一个字节都需要请求 CPU 中断当前的程序而为之服务(如 8251，8255)。8089 的引入，则大大减轻了 CPU 对外设的负担。当 CPU 需要进行 I/O 操作时，它只要在存贮器中建立一个信息块，将所需要执行的操作，有关的参数按照规定列入，然后通知 8089 前来读取。

8089 读得操作控制信息后，能自动地完成全部的 I/O 操作。因此，对一个配接有 8089 的系统的 CPU 来说，在所有的输入输出操作中，数据都是整块地成批发送或接收，而把一块数据按字或字节同 I/O 外设交换(如 CRT 终端，行式打印机等)都是由 8089 来具体完成的，这一切对于 CPU 来说都是"透明的"，CPU 在这个时候可以并行地执行某些其他操作。不难看出，当 8089 配接到 iAPX 86/88 系列的 CPU 上以后，它就承担了原来需要由 CPU 完成的输入/输出操作，有效地减少了 CPU 在 I/O 处理中需要的开销，因而从另一个侧面提高了 iAPX 86/88 组成的系统的性能。关于 8089 的详细情况可参见第九章。

三、操作系统固件 80130 (OSF)

为了满足 iAPX 86/88 系列用于实时、多任务系统的需要，减轻用户在设计实用操作系统时的困难，Intel 公司又推出了 OS 固件 80130。当它与基本的 8086/8088 配接后，就能构成一个完整的操作系统处理机 OSP。

OSP 以固件形式为用户提供一个完好定义的、并已调试完成的多任务操作系统原型。该固化的操作系统能支持 5 种数据类型：

Job 型，Task 型，Segment 型，Mailbox 型和 Region 型。

固化的操作系统还能提供 35 种基本命令，以实现如下功能：

(1) 作业(Job)管理和任务(Task)管理

(2) 中断处理

(3) 空白存贮单元管理

(4) 任务间通讯

(5) 任务间同步

(6) 环境控制

例如，当 OSP 在执行任务生成，延迟，挂起以及执行操作时，对任务的状态，包括寄存器状态和软件控制信息都进行管理。又如，OSP 能支持面向事件的系统设计。此时每个事件可相对地设计成一个任务，并将相关的任务放在一个公共任务中，当事件发生后，启动片内的中断控制机构，从而进行实时处理。OSP 片内的中断控制逻辑可以从 8 个扩充到 57 个。

此外，80130 还能配接到带有协处理器(如 8087)的系统中，它的接口还能与多总线系统相兼容，关于 80130 的详细情况可参见第十一章。

§8.2.3 iAPX 86/88 系列性能的纵向提升

iAPX 86/88 系列经过 8087，8089，80130 等协处理器的横向性能提升，充分发挥了 8086/8088 的潜力，但由于受到 8086 基本结构的影响，在处理器的运算速度和功能上不可避免地还

带着早期16位微处理器的缺陷，特别是缺乏虚存和保护机构，使iAPX 86/88系列在多用户、大容量的应用面前有些捉襟见肘。针对这种情况，Intel公司又采用了对iAPX 86/88系列的性能进行纵向提升的策略。一方面使新一代的16位微处理器保持与原有的产品相兼容，另一方面又对处理器的性能进行了大幅度的提高。Intel公司先后推出了80186/80188和80286（即iAPX 186/188和iAPX 286）。80186是在增强了的8086 CPU的基础上，集成了许多外围部件，从而可使系统结构大大简化，同时能减少大量的总线操作。80286则把16位微处理器的性能提高到了一个新的水平，它一方面保留了8086的基本结构及全部指令，以便与原来的产品兼容，另一方面又增加了存贮器管理和虚地址保护机构。80286的问世，标志着16位微处理器进入了一个新的时代。iAPX86，iAPX186与iAPX286的各种性能参数见表8.4

表8.4 iAPX 86，iAPX 186，iAPX 286性能比较

	iAPX 86	iAPX 186	iAPX 286
寻址能力	1 M 字节	1 M 字节	16M（虚拟1000M）字节
分 段	4×64 K	4×64 K	(1～16K)×64K
时 钟	标准5MHz (有8MHz、10MHz)	标准8MHz	标准8MHz (有10MHz)
组 成	8086	8086 (8MHz) 时钟发生器、计时器、 DMA通道，中断控制器 CS/READY逻辑	集成化存贮管理 多任务支持 4级保护
软 件	PL/M, ASM, PASCAL FORTRAN, ICE-86	PL/M, ASM, PASCAL FORTRAM, ICE-186	PL/M, ASM, PASCAL FORTRAN, ICE-286 RMX-286
片性能	1×8086	2×标准8086	6×标准8086

一、高性能的16位微处理器80186

在许多不同的8086系统设计中，使用了不少相同的部件来完成一个系统所必须的功能，例如，需要一个为多重外部中断提供中断类型数的中断控制器，需要提供系统时钟的时钟发生器，需要进行快速输入/输出的DMA控制器等。在构成系统时，这些器件要占用很大的印刷板空间，器件之间的互连也往往会成为设计者头痛的问题。如果能把构成一个系统所必需的器件组织在一块芯片上，就能大大简化系统的设计，并能提高系统的可靠性和速度。80186/80188正是这样的CPU芯片，它们是和8086/8088相兼容的。其中80188除了在外部只有8根数据线以外，基本结构和80186相同。80186/80188在一块68脚的芯片上集成了下列硬件：

(1) 8 MHz的增强的8086 CPU
(2) 一个可编程序的中断控制器部件
(3) 一个时钟发生器部件
(4) 一个片选和准备就绪信号发生部件
(5) 一个具有3个计时器的计时器部件
(6) 一个具有2个通道的DMA控制部件

这些部件的控制信息可以用程序的方法在输入/输出空间或存贮器空间中寻址。80186比8086还增加了10条新的指令。由此可见，80186/80188的出现适应了当前用16位微处理器构成更大系统的需要。关于80186/80188的详细情况可参见第十四章。

二、超级 16 位微处理器 80286

80286 是新一代的 16 位微处理器，它革除了 Intel 产品以前一直沿用的多重地址/数据总线而代之以分立的地址、数据线。80286 除了能与 8086/8088、80186/80188 相兼容外，还具有许多原来 iAPX 86/88 产品所不具有的新的特性和功能。80286 在一块 68 脚的芯片上除了集成有 8086 的基本结构外还集成了存贮器管理和虚地址保护机构。10 MHz 的 80286 的整体功能比 5 MHz 的 8086 提高了 6 倍。

80286 能以两种方式进行工作，即实地址方式和保护虚地址方式。用实地址方式构成的系统相当于 8086 的最大模式，在这种方式下，80286 的寻址能力为 1 兆字节。除了能以更快的速度执行以外和 8086 的最大模式系统基本相同。80286 的实地址方式能方便地转换为保护虚地址方式，从而使它与以前的 Intel 产品有着良好的兼容性。

在保护虚地址方式下，80286 能寻址 16 兆字节，能支持多任务操作，并能为每个任务提供多达 1000 兆字节的虚地址空间，这样的地址空间，对一般的应用就几乎可以认为是"无限"的了。80286 集成有一系列的硬件来支持虚、实地址的转换，80286 还提供了四层特权，并有特权检查机构来实现任务和操作系统，任务和任务之间的相互隔离。80286 提供 15 条新的保护控制类高级指令来支持这些操作。象 8086 一样，80286 也能进行横向的性能提升，例如加接数值数据协处理器 80287 以后，就能大大提高 80286 的数值数据处理速度和能力。80286 的这些卓越的性能已经接近或超过了某些典型的小型机，是目前最强的 16 位微处理器。

为了方便用户使用 80286 构成系统，Intel 公司推出了以 80286 为 CPU 的一整套主机和系统扩充板，即 iSBC286/10 产品系列。这个系列的核心就是 iSBC 286/10 单板计算机，它受到 iRMX™ 286R 操作系统的支持因而受到各方面的关注。

最近，Intel 又推出了以 80286 为核心的 OEM 16 位微机系统 SYS 310-5，由于这是一个采用 XENIX 操作系统的多用户 16 位微机系统，所以有着十分诱人的前景。关于 80286 和 iSBC 286/10 产品系列的详细介绍可参见第十五章和第十六章。

§8.3 iAPX 86/88 系列的总结

以上我们对 iAPX 86/88 系列的形成和发展作了分析介绍，从中可以看出 16 位微处理器发展的历程和目前所达到的水平。正因为 iAPX86/88 系列是一个如此完整的 16 位微处理器系列，所以才如此受到各方面的重视。

为了帮助读者全面地掌握 16 位微处理器，我们在以下各章中对使 8086 微处理器的性能得到提升的各个方面以软件和硬件相结合的方式进行论述。其先后次序为：横向的性能提升，着重介绍了 8089，8087 和 80130；对 8 位机性能进行提升的 8088，及一些重要的外围芯片；8086 纵向的性能提升，着重对 80186/80188，80286 及 iSBC 286/10 产品序列进行了论述。其中章与章之间并没有什么依赖关系，读者可以顺着次序读下去，以求掌握 16 位微型机发展的全貌，也可以分章地根据需要进行选读。由于各种有关的资料浩如烟海，我们只能挑出最主要的部分经过消化与分析后介绍给读者。中册是以上册为基础的，故我们假定读者是已经掌握了 8086 的结构和指令系统的。如果读者对此还感到生疏，可以先参阅本书上册的有关章节。同样对在上册中解释过的名词，中册里一般不再解释。

第九章　8089 输入/输出处理器

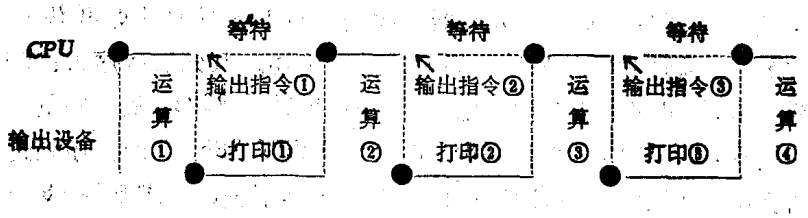
§9.1　8089 I/O 处理器概述

在计算机的发展史上，输入输出曾经是计算机系统的"瓶颈"，即便是在今天的不少系统中，这一问题仍然严重地存在着，追根求源，之所以会出现这种现象，主要是由于输入输出的工作速度取决于机械动作，它与计算机的内部处理速度（电子的运动）相比就显得太慢了。例如，对纸卡片读出装置来讲，它以每分钟 800 张的速度读纸卡片，对每个字符的处理需要一毫秒的时间，而对中央处理机来讲，就连比较花费时间的 36 位除法也能在这一毫秒内执行约 100 次。可见速度差别之大。正因为如此，人们为设计出"好"的输入输出系统，缓和这些矛盾而做了很多工作，取得了不少的成功经验。

我们知道，最初的计算机是按单用户设计的，输入输出（I/O）系统的设计任务主要是要解决好运算器、主存及 I/O 设备在速度上的巨大差距，那时的输入输出过程是由程序员自己安排的。随着分时系统的出现，每个用户程序的输入和其运算结果的输出当然就不再能够由各用户自己安排，而要由输入输出系统来专门管理。这样一来，I/O 系统就相当重要，其性能的好坏不仅要影响到各个用户从程序输入到结果输出所占的时间，而且还会影响到 CPU 和主存利用率的高低。随着计算机系统的不断发展和应用领域的进一步扩大，系统所要求的输入输出数据量迅速增大，输入输出设备的种类也日益增多，从而导致输入输出系统的优劣直接影响到计算机系统的性能。下面我们来简单地回顾一下计算机输入输出系统的发展过程，大致上可分为三个阶段：

一、程序控制输入输出的方式

大家知道，早期的计算机是以 CPU 为中心的，外部设备由 CPU 直接管理，也就是说，输入输出操作完全由 CPU 控制，由 CPU 启动、控制、停止输入输出过程。输入输出设备与主存之间的信息交换需要经过运算器。在进行输入输出操作期间，停止 CPU 的动作，直至数据交换完毕。其过程可表示为：



为了使用某个外部设备，CPU 需要选定和指明所要用的设备；CPU 需发出控制信号使该设备与运算器相连；CPU 需发出该设备能"理解"的控制命令去启动该设备，使它能接收（对应于输出操作）或发送（对应于输入操作）信息；该设备还需能在接收或发送一个信息元（如一个字符）时，发送给 CPU 相应的控制（或状态）信息；由于信息的输入输出一般是成块进行的，因而，CPU 还需能检测或判定信息块的终点；又因为输入、输出设备的信息表示（如 2—10 进制）

可能和CPU内的信息表示(多为2进制)不同，以及输入、输出设备与CPU之间的传送方式(如串行传送与并行传送、字节传送与字传送)和CPU与主存之间的传送方式(都是并行的字节或字传送)的不同，又要求CPU适当地进行信息的变换、装配或拆卸。此外，CPU还应该能够递增主存中输入、输出信息块区域的地址，同时还要递减信息块长度计数器的值。所有这些操作都是使用I/O设备时所必须的，而I/O系统结构发展中的一个主要方面就是如何合理分配这些操作，使它们分布实现，而不是如程序控制输入输出的方式那样全由CPU完成。从后面的介绍中就可以看出来，设计8089的目的就是让它从CPU接管这些任务。

在程序控制的输入输出方式中，每输入输出一个信息元就要循环执行一段控制程序，更由于它要等待来自慢速的I/O设备的状态信息，致使计算机系统的使用效率十分低劣，资源得不到充分的利用，CPU的高速运算能力远远得不到发挥。

中断概念引入输入输出过程是对这一方式的一大改进。例如，假设程序要进行打印，CPU执行一条打印(输出)指令，在启动了打印机之后，CPU可不必空等由打印机返回的状态信息，而转去执行别的操作。在打印机做好准备时，它就发出中断请求，CPU响应该中断，并将信息送入打印机，接着，CPU又可继续执行自己的工作。这样一来，CPU与输入输出设备之间就能在一定程度上并行工作。同时，由于启动一种设备到该设备发出中断请求之间的时间较长，足以启动别的设备，从而就可以实现多种外部设备的同时工作。但是，这种方式的效率也是不高的，其原因就在于：每传送一个信息元，CPU都得执行一段中断处理程序。随着外部设备种类和数量的增多，会造成中断次数过于频繁，从而耗费掉大量的CPU时间。

二、DMA(直接存贮器访问)方式

在程序控制输入输出的方式当中，信息是从内存经运算器与外部设备交换的。DMA方式打破了这一格局，它在主存与外部设备之间设置了直接通路及相应的控制逻辑，使外部设备能直接与主存相连，从而显著减轻了CPU的负担。

一般说来，DMA内有数据缓冲寄存器、设备地址寄存器、主存地址寄存器、数据计数器以及其它一些控制线路。CPU在开始进行数据传送之前，先要给这些寄存器装入适当的信息，即把设备号送至设备地址寄存器，并启动该设备；同时，还要将用于数据传送的主存区域的起始地址置入主存地址寄存器，并将所需传送的次数送入计数器。在此之后，CPU就可以继续转去执行其它程序。

对于DMA成组传送方式来说，当输入输出设备准备好接收或发送数据时，它就对CPU发出DMA请求，使输入输出设备直接与主存交换数据。每传送一个数据，主存地址寄存器内的地址就加"1"，数据计数器则减"1"。当数据计数器的内容为"0"时，说明本次DMA传送结束，输入输出设备给CPU发一中断信号。采用DMA方式，有可能在一主存周期内完成一个数据的传送。

三 I/O处理机方式

引进了DMA方式之后，输入输出系统分担了输入输出过程中的部分操作，确实简化了CPU对输入输出的控制。但是，也应该看到，对输入输出设备的管理和不少操作仍需CPU承担，象信息的变换、装配、拆卸和数码校验这些功能都得CPU实现。为了使CPU摆脱用于管理、控制I/O系统的沉重负担，在六十年代初期就引进了I/O处理机的概念，出现了第三种输

入输出方式——I/O 处理机方式.

I/O 处理机几乎把控制输入输出操作及其信息传送的所有功能，从 CPU 那里接管过来，独立出去。I/O 处理机有自己的指令系统，也能构成和执行程序。它能对外部设备进行控制对输入输出过程进行管理，还能完成码制变换、字与字节之间的装配与拆卸、整个数据块的错误检测和纠错、格式变换等运算和操作。此外，它还可以向 CPU 报告设备和设备控制器的状态，对状态进行分析，并能对输入输出系统出现的各种情况进行处理。所有这些动作都可与 CPU 程序并行进行。

与 I/O 处理机类似，通道也是一种管理输入输出的"装置"，也可以把它看成是"处理机"，它也有指令，并能构成通道程序。我们可把通道方式归入 I/O 处理机方式之内。

由上面的讨论可以看出，要想使 CPU 的处理与输入、输出动作并行进行，首先必须使外部设备在任何时刻都能独立地工作；其次，就是要求让外部设备工作所需要的各种控制和信号，与中央处理机无关，而是由外设控制器自动地独立形成。

更具体地讲，在微型计算机飞速发展的十余年间，其输入输出系统又是如何发展起来的呢，即 CPU 与 I/O 设备之间的接口关系的演变过程是怎样的呢？

近年来，由于半导体技术和集成电路工艺的迅速发展，计算机技术突飞猛进，微型机在其短短的十年发展历史中已经历了三代更新。第一代微型机是以 Intel 的 8008 为代表的；第二代的代表机型有 Intel 公司的 8080/8085、莫托洛拉公司的 M 6800、泽洛克公司的 Z 80 等；至于 Intel 公司的 iAPX 86/88 及其改进型 iAPX 286/186、莫托洛拉的 M 6800 及泽洛克的 Z 8000 则属于第三代微机产品。图 9.1 以 Intel 公司为例，说明了 CPU 的演变过程。对应于 CPU 的三代产品，我们也可以把微机中 CPU 与 I/O 设备之间的接口关系的发展分成三代。第一代的 I/O 接口是由 TTL 门电路所组成的，即采用的是 TTL 随机逻辑控制方式，用这种方式管理输入输出时，需要 CPU 对外设的每一个输入输出电平进行控制。第二代的 I/O 接口是一些专用的外围接口芯片，例如通用的并行接口 8255、串行接口 8251 以及 DMA 控制器 8257 等芯片，这些接口控制芯片要比 TTL 随机逻辑控制方式优越得多，但其输入输出工作仍然在

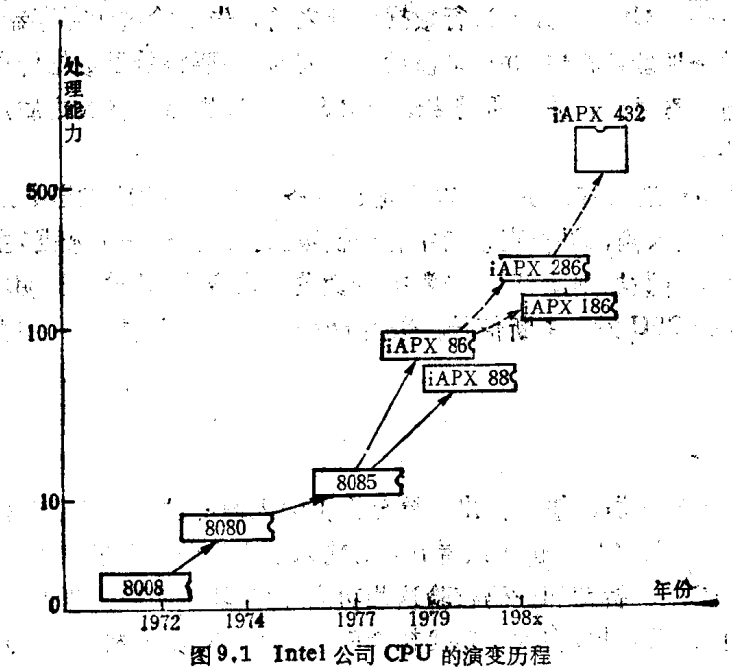


图 9.1 Intel 公司 CPU 的演变历程

相当程度上需要依靠 CPU 的支持，它几乎每读入/发送一个数据元都需要请求 CPU 中断当前的程序而为之服务。例如，8251 每接收或发送一个字节都要向 CPU 发出中断请求。显然，采用这种方式，在输入输出频繁的情况下，CPU 的 I/O 开销仍是相当可观的。8089 使微处理机的 I/O 处理进入了第三代，作为输入输出处理器，8089 大大减轻了 CPU 管理输入输出工作的负担，有效地担当了原来需要由 CPU 完成的输入输出操作，从而有效地减少了 CPU 在 I/O 处理中所需要的开销。

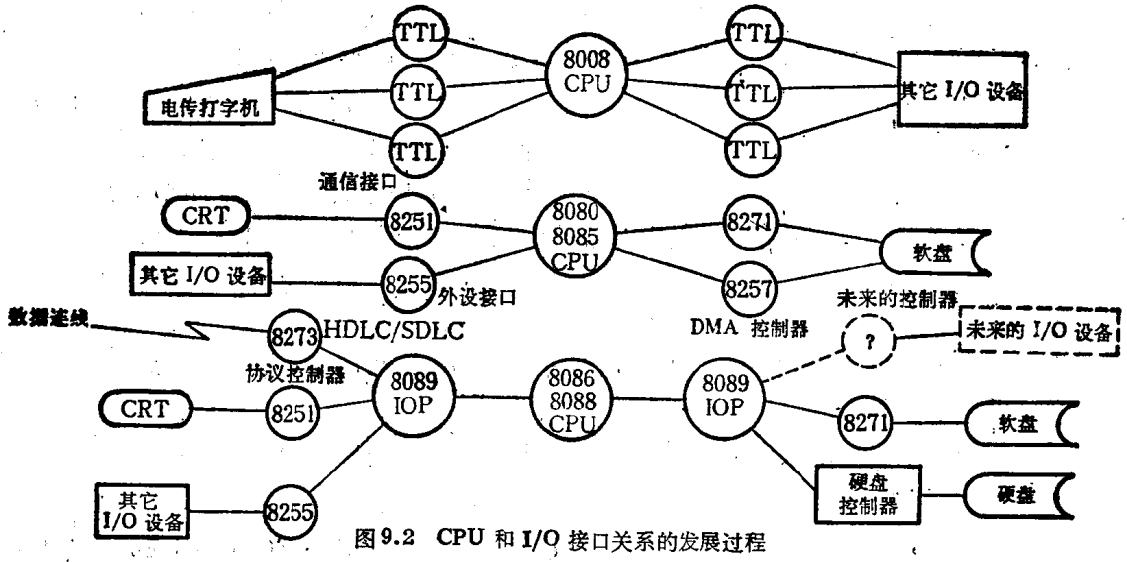


图 9.2 CPU 和 I/O 接口关系的发展过程

Intel 8089 给微型计算机系统的输入输出处理带来了革命性的变化，它将目前只在大型计算机中才普遍使用的 DMA、通道和 I/O 处理机的概念引进到微型计算机的领域当中，为微机的输入输出系统之设计开创了一条新路子。

8089 是为 Intel iAPX 86/88 系列所设计的一个智能 DMA 控制器，它具有较强的处理能力。当把它接到 iAPX 86/88 系列的 CPU 上以后，它就能有效地完成原需 CPU 完成的输入输出操作，从而能大大降低(几乎去除了) CPU 在输入输出处理中所需要的开销，实际上这也就从一个侧面提高了由 iAPX 86/88 所组成系统的性能。在加了 8089 的系统中，当 CPU 需要进行 I/O 操作时，它只要在存贮器中建立一个信息块，将所需要执行的操作、有关的参数按照规定设置好，然后通知 8089 前来读取这些信息。8089 I/O 处理器(也可简称为 IOP)读取这些信息，负责执行全部 I/O 操作，执行完成之后再通知 CPU。而对于 CPU 来讲，所有输入/输出操作中的数据都是用 DMA 方式成块地接收或发送的，至于将一块数据按字或字节分次与I/O 设备交换(如与 CRT 终端、行式打印机等)这样一些具体的工作则由 8089 完成，也就是说由 8089 去进行和外部设备之间的数据传送，由 8089 承担一切的时间开销。如果发现传送结果中出现差错，8089 还可以决定重复进行传送，并可进行某些处理。在 8089 进行操作的过程中，CPU 不必去干预它的工作。因此，CPU 可以和 8089 并行地去处理其它工作。由此可见，在输入/输出要求很高时，使用 8089 能够显著提高整个系统的性能。

8089 是一种单片的通用输入输出处理器，它包括两个“独立的” DMA 输入/输出通道，每个通道都有自己的寄存器组，根据各自的优先权，这两个通道可以交替地工作，当标准的时钟频率为 5 MHz 时，每个通道的传输速度最高可达 1.25 兆字节/秒。IOP 与 CPU 之间以存贮器为基础的通讯设计方法提高了系统的灵活性，有利于软件的模块化设计，从而为产生更可靠、

更易发展的系统提供了可能性。

8089 把微处理器的处理功能和直接存贮器存取(DMA) 控制器结合了起来，它除了具有通常的 DMA 传送功能之外，还能够在传送数据的同时，对数据进行翻译和比较、装配和拆卸，并可以设置多种结束数据传送的条件等等。8089 的指令集及性能都经过优化处理，以满足高速、灵活及高效的 I/O 处理这样一些要求。利用 8089，还为 16位的 8086 和 8 位的 8088 与 8 位和 16 位外设之间的相互连接提供了方便。

8089 采用的是 40 条引脚的双列直插式封装，电源为单一的 +5 伏电源。它与工作在最大模式中的 8086 和 8088 直接兼容，同时，它也和 Intel 的多总线结构相容。

8089 与 iAPX 86/88 系列 CPU 的结构方式有两种：一种叫本地(或局部) (LOCAL)方式；一种叫远程(REMOTE)方式。采用 LOCAL 方式时，8089 与 CPU 共享系统总线，即可在不增加任何硬件的情况下，就能实现两个 DMA 通道的功能，这种方式在一般的小型系统中是比较经济实用的。远程方式则是一种比较高效的连接方式，在采用 REMOTE 方式时，8089 与 CPU 共享系统总线，但 8089 又有自己的 I/O 总线，这样，8089 不必通过系统总线就能访问它自己的 I/O 外设和局部存贮器。远程方式能够减少 8089 与 CPU 争用总线的次数，提高了 8089 与 CPU 并行工作的程度，同时，采用这种结构时，可以在系统中连接多个 8089 子系统。

归纳起来讲，8089 具有如下特征：

1. 高速 DMA 能力。包括 I/O 设备到存贮器、存贮器到 I/O 设备、存贮器到存贮器、I/O 设备到 I/O 设备之间的 DMA 传输。

2. 与 iAPX 86/88 系列的 CPU 兼容。在 iAPX 86/11 (一片 8086 加一片 8089 组成的系统)或 88/11 (8088 CPU 加上 8089 组成的系统)结构当中，能够大大降低 CPU 的输入输出开销。

3. 使 8 位及 16 位外设与 8 位及 16 位处理机总线之间的各种连接成为现实。

4. 1M 字节的寻址能力。

5. 与 CPU 之间的通讯是以存贮器为基础的。

6. 提供了局部(LOCAL)和远程(REMOTE)这两种 I/O 工作方式。

7. 灵活的、智能的 DMA 功能，包括翻译、检索、字的装配/拆卸功能。

8. 与多总线(MULTIBUS)兼容的系统接口。

§9.2 8089 的工作原理

由于 8089 是一个引进了许多新思想、新概念的 I/O处理器，所以为了便于读者掌握，我们准备在本节中先简单地介绍一下 8089 的基本操作过程，建立一些新的基本概念，以此作为本章的基础。在本节的概述中，有意地省略了一些操作细节，对于这些细节的完整说明将在本章的后面几节中详述。

§9.2.1 CPU 与 8089 之间的通信

一般说来，CPU 与 IOP 之间的通信是借助于存贮器进行的，通过两者共享在存贮器中准备的一些信息而间接地实现相互之间的通信。我们可以把 CPU 与 IOP 之间的通信方式分为

两种类型：初始化方式和命令方式。

初始化通常是在系统加电或复位后进行的。CPU 在内存中为 IOP 建立一个信息区，然后通知 IOP 去读取这些信息。利用这些信息，8089 就可以确定它应如何去进行输入/输出操作，确定出系统总线和 I/O 总线的宽度（8 位或 16位）以及它应以哪种模式去工作，也只有这样，8089 才能够知道如何去控制、访问总线。实际上，在一般的工作过程中，8089 好象是两个独立的器件——通道 1 和通道 2。CPU 与通道之间的通信全部集中于图 9.3 中所示的通道控制块(CB)上，通道控制块位于系统存贮空间中，CPU 在进行 IOP 的初始化处理时，也就把 CB 的地址告诉了 IOP，一个通道控制块专用于某个 IOP 的两个通道。CB 中的忙碌标志(BUSY)表示 IOP 正在进行操作，还是在等待 CPU 发给它新的命令。CPU 通过通道命令字 CCW 告诉 IOP 应当进行哪一种操作。

当 CPU 命令某个通道去执行一段程序时，IOP 就要使用图 9.3 中所示的参数块和任务块。参数块类似于程序送给子程序的参数表，它里面所存放的是执行通道程序时所要用的变量数据，参数块中还为通道返回 CPU 的变量(操作结果)保留了空间。参数块在 IOP 与 CPU 之间起着一个信息中心的作用，在参数块中传递 CPU 到 IOP 的参数或可变信息，借助于它，还能在 IOP 与 CPU 之间传输数据和状态信息。此外，参数块中还包含有任务块的地址。

任务块就是用 8089 指令写成的通道程序，利用它来执行所需要的操作。例如，它可以利用参数块的数据去建立 IOP 及外设控制器的初始状态，进行数据传送，返回结果，然后处于待命状态。任务块既可以放在系统存贮器当中，也能够放在 IOP 的局部存贮器当中，以允许 IOP 与 CPU 并行运行。

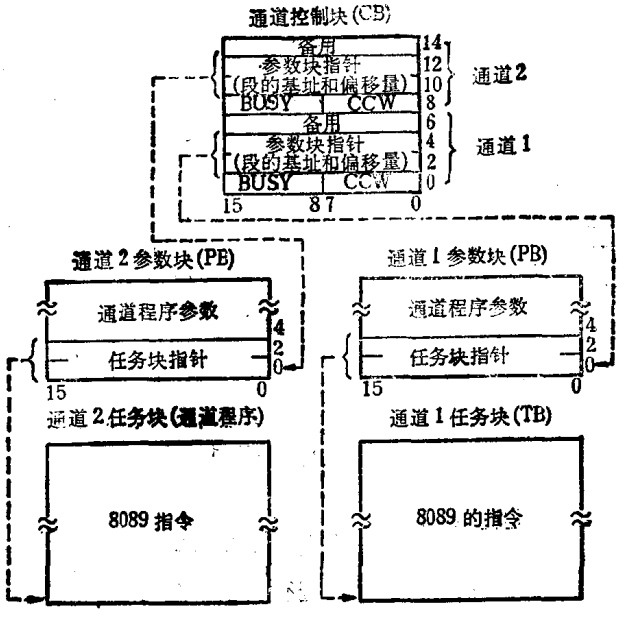
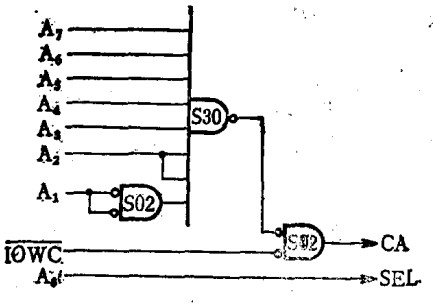


图 9.3 CPU 和 IOP 之间的通信机构

在 CPU 命令 IOP 执行通道程序之前，它先得把任务块(TB)、参数块(PB)和控制块(CB)按图 9.3 所示的格式链接起来。链接的方法是利用 8086/8088 的标准双字指针变量，即低地址的字代表偏移量，高地址的字代表段的基地址。在一个系统中可以有许多用途不同的参数块和任务块，但在任何时刻只允许一对参数块和任务块与某个通道相连。在 CPU 填上 CCW 并把 CB 连接到某个 TB 及 PB 之后，在条件允许时，就可以发出通道注意信号，由于 IOP 有两个

通道，所以还要在两个通道中间作选择，一般都是在通道注意(CA)信号的后沿利用地址线 A_0（连到 IOP 的SEL 引脚)来选择所需要的通道(通道 1 或通道 2)。这一过程如图 9.4 所示。因此，如果 IOP 按照 CPU 的 I/O 转接口编址，它的两个通道就相当于两个相邻的 I/O 转接口；若按存贮器进行编址，它们就相当于两个连续的存贮器单元的地址，此时，访问这些单元的指令(如 MOV)就将产生通道注意信号。



I/O 转接口地址若为 FCH，则代表通道 1 的 CA 信号
I/O 转接口地址若为 FDH，则产生通道 2 的 CA 信号

图 9.4　通道注意信号的译码电路

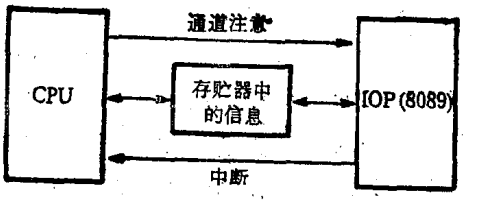


图 9.5　CPU 与 IOP 之间的通信

IOP 通道注意信号的作用类似于 CPU 的中断。IOP 在收到通道注意命令时，就停止它的当前工作去查看 CCW 中的命令内容，倘若是一条开始执行通道程序的命令，它就把参数块和任务块的地址装入内部寄存器，把 BUSY 标志置 1，然后开始执行通道程序。而 CPU 在发出 CA 命令之后，就可以自由地去处理其它任务。通道和 CPU 可以同时进行不同的操作，各干各的事情。当通道执行完了通道程序之后，就把 CB 中的 BUSY 标志位清 0，并以此作为向 CPU 报告的信息。有时候，IOP 也可以向 CPU 发出中断请求信号，以表示通道程序已经执行完毕或某个事件已经发生。总之，CPU 与 IOP 之间的通信可以用图 9.5 来表示。由图可见，除了通道注意 (CA) 和中断请求信号之外，CPU 与IOP 之间的通信都是通过共享存贮器中的信息来实现的。

8089 的两个通道之间是彼此独立的，它们都有自己的寄存器组、通道注意信号输入端、中断请求输出端以及 DMA 控制信号。在每个瞬时，通道可能处于对通道注意信号进行响应处理、执行 DMA 传送、执行通道程序或处于空闲待命这样一些状态。两个通道可以交替进行工作，但在任一时刻，最多只能有一个通道在进行实际的操作。机内的优先权系统使某个通道上优先级别较高的活动能够抢在另一通道上的“不太重要”的操作之前进行。当然，CPU 还可以调整优先级别，以处理某些特殊事件。CPU 也可以通过不同的 CCW 命令来启动通道、停止通道的工作或挂起通道的操作。

通道程序是用 8089 的汇编语言 ASM-89 编写的，ASM-89 包括 50 多条基本指令。这些指令可以对位、字节、字、双字(指针)等变量进行操作，也可以对 20 位的实际地址进行处理，这是 8086 和 8088 中所没有的。数据可以来自寄存器、存贮器，也可以用立即数作为操作数。8089 提供了四种寻址方式，它们可以灵活地对 CPU 的一兆字节的存贮空间和 8089 的 64 K 字节的 I/O 空间进行寻址。

8089 的指令系统包括与 CPU 中类似的一些通用指令，也含有一些专用于 I/O 操作的指令。其中包括数据传送指令、简单的算术逻辑操作指令、地址处理指令、控制转移指令、位操作指令以及启动 DMA 传送的指令等。

§9.2.2 8089的DMA传送

8089的XFER指令专用来使通道准备进行DMA传送。在XFER指令之后再执行任意一条指令，都将使8089停止执行程序，进入DMA传送方式。因为DMA传送是在通道寄存器的管理之下进行的，所以，在执行XFER指令之前，必须把通道寄存器的内容预先设置好。

数据从源送至目标。源和目标可以是CPU的存贮空间中的任一存贮单元，也可以是8089的I/O空间的任一单元，8089对这两种存贮空间是不加区别的，因此，数据传送可以在I/O接口和存贮器、存贮器和I/O接口、存贮器和存贮器、I/O接口和I/O接口之间进行。至于它们之间8位和16位的关系则由8089负责协调处理。

任何一个传送周期都可以根据事先的约定，利用来自源或目标的DMA请求信号实现同步，即在同步方式中，通道要等同步信号到来之后才开始下一个传送周期。当然，传送也能以异步方式进行，在这种情况下，通道在前一周期结束后立即开始下一个传送周期。每个传送周期都是分两步实现的：先从源取出一个字节或字送入8089，然后再由8089把该数据元存入目标单元。8089能够自动地以最有效的方式利用系统中的数据总线，例如，在把8位外设中的数据送到16位总线的存贮器中去（如8086的存贮器）时，8089就将自动地执行两次读数据（字节）周期，读进一个整字，然后用一个周期把这个整字存入目标。这样，8089就利用三个总线周期完成了一次字传送，而不是用四个总线周期（两个读字节周期，两个写字节周期）来实现这一传送功能。

在取周期和存周期之间，8089可以对数据进行加工处理，比如说，可以把该字节由EBCDIC码翻译成ASCII码，也可以把它与某个要查找的数值进行比较等等。

程序员可以用多种方法来结束DMA传送，例如，可以规定在传送了某个指定的字节数目之后停止传送，注意，其中的字节数目不能超过64K；也可以在通道的结束传送引脚上加上一个传送结束信号，以表示当前DMA传送结束，这样的结束信号大多数都是由外部设备发出的；通道还可以将传送中的字节数据与某个给定字节数据相比较，根据比较的结果（相符或不相符）决定是否结束DMA传送；还有一种所谓的单周期传送，即每传送一个字节或一个字就终止DMA过程。

在DMA传送结束之后，通道就可以自动地恢复执行通道程序，而通道程序又能够规定结束DMA传送的条件，例如，它可以规定在传送了80个字节或遇到回车字符时，结束后面的DMA传送过程。通道程序还可以对刚结束的传送进行一些处理，比如说，它可以从结果寄存器了解到刚结束的传送是否成功，倘若不成功，通道程序能够在不需CPU介入的条件下要求通道再传送一遍。

通道程序在结束时，常常都把操作结果送到参数块中，有时也向CPU发出中断请求信号，然后通道进入暂停（HALT）状态，同时清除通道控制块中相应的忙碌（BUSY）标志，表示该通道又可以接收新的命令，去完成一项新的任务了。当CPU被中断时，它还可以通过检查该BUSY标志以确定操作是否已经结束。

§9.2.3 总线和系统结构

8089能够访问1兆字节的系统空间和64K字节的I/O空间，尽管8089实际上只有一条数

据总线，但若把它想象成包括一条访问系统空间的系统总线和一条访问 I/O 空间的 I/O 数据总线，则往往是非常有用的。利用总线控制器 8288 输出的周期类型信号，可以从逻辑上区分开这两类总线，位于系统空间中的存贮单元或 I/O 转接口只能通过存贮器读写信号访问，而位于 I/O 空间的存贮单元或 I/O 设备只有用 I/O 读写信号才可访问。换句话说，系统空间中的 I/O 转接口是按存贮器编址的，而 I/O 空间中的存贮器则是按 I/O 进行编址的。

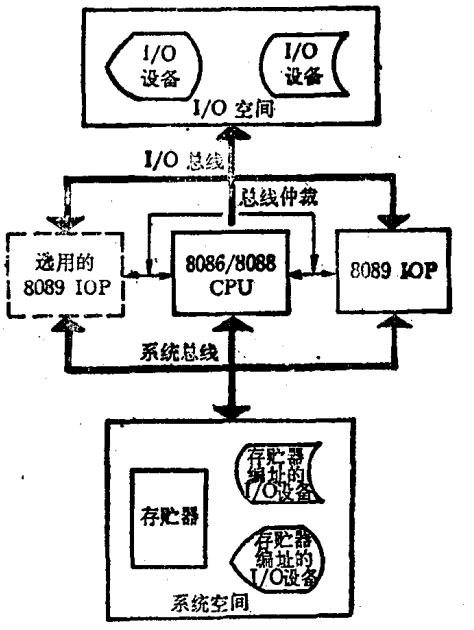


图 9.6　8089 的本地方式结构示意图

在前一节中，我们已经提到过 8089 的两种系统结构方式，也就是本地方式和远程方式。本地方式的最主要的标志就是 8089 与 CPU(8086 或 8088)共享系统总线和 I/O 总线。在本地方式的系统结构中，系统总线的宽度与 I/O 总线的宽度是相同的，也就是说，若 CPU 是 8088，则总线的宽度均为 8 位；若 CPU 为 8086，则其总线宽度都应该是 16 位的。此外，在本地方式中，8089 的系统空间及 I/O 空间与 CPU 的对应空间也是一样的，通道程序存放在系统空间中，但 I/O 转接口既可在系统空间中，也可以在 I/O 空间内。在这样的系统当中，尽管 8089 可以与 CPU 公用总线，但不允许它们同时使用总线。本地方式这种结构的优点是，除了增加 8089 之外，不需要增加其它元件，就可以获得具有一定智能的 DMA 和一组功能很强的 I/O 指令。而缺点就在于 8089 和 CPU 的并行操作受到不能同时使用总线的限制。上图给出了本地方式的基本结构。在本地方式中，8089 是作为工作在最大模式下的 8086（或 8088)的一个从设备出现的。图 9.7 更具体地表示出了本地方式中，8089 和 CPU 共享系统的地址锁存器(8282)、双向数据驱动器(8286)及总线控制器 (8288) 的情况。从图中可以看出，8089 和 CPU 使用总线是按照请求/批准($\overline{RQ}/\overline{GT}$)规则进行的，从设备 8089 可以向 CPU 请求使用总线，CPU 在自己不占用总线时，就可以把总线控制权让给 8089，然后等待 8089 用完总线。在 8089 用完总线之后。它就放弃总线控制权，CPU 又重新获得了总线的控制权。由于 CPU 是总线的主控者，故在 8089 占用总线时，CPU 不可以请求使用总线。

在 8089 的另一种系统结构——远程方式当中，系统总线是由 8089 和 CPU 共享的，但 8089 有自己的 I/O 总线，这样就使得 8089 和 CPU 可以并行操作而不会发生使用总线的冲突。在这里，系统总线的共享实际上是分时共享，也就是说，8089 和 CPU 不可以同时使用系统总线，总线归谁使用这一决策要由总线裁决器 8289 作出。在如图 9.8 所示的远程方式的系统结构当中，我们可以看出 8089 的 I/O 总线和 CPU 的系统总线在物理上是分开的。正因为如此，就不再需要 I/O 总线的宽度与系统总线的宽度相同了。举例来说，假设 8089 与 CPU 之间的通信(系统总线)采用的是 16 位的数据总线，那么，8089 与 I/O 设备之间的通信(I/O 总线)既可以采用 8 位的数据总线，也可以采用 16 位的数据总线。在远程方式当中，8089 与本地 I/O 转接口之间的通信不需使用系统总线；倘若把通道程序存放在本地 I/O 空间当中，则 8089 即便是取指令也不需要通过系统总线；但必须把参数块、通道控制块以及后面将要介绍的系统结构块(SCB) 等 CPU/8089 通信所用到的模块放在系统空间中，从而使 CPU 与 8089 都能访

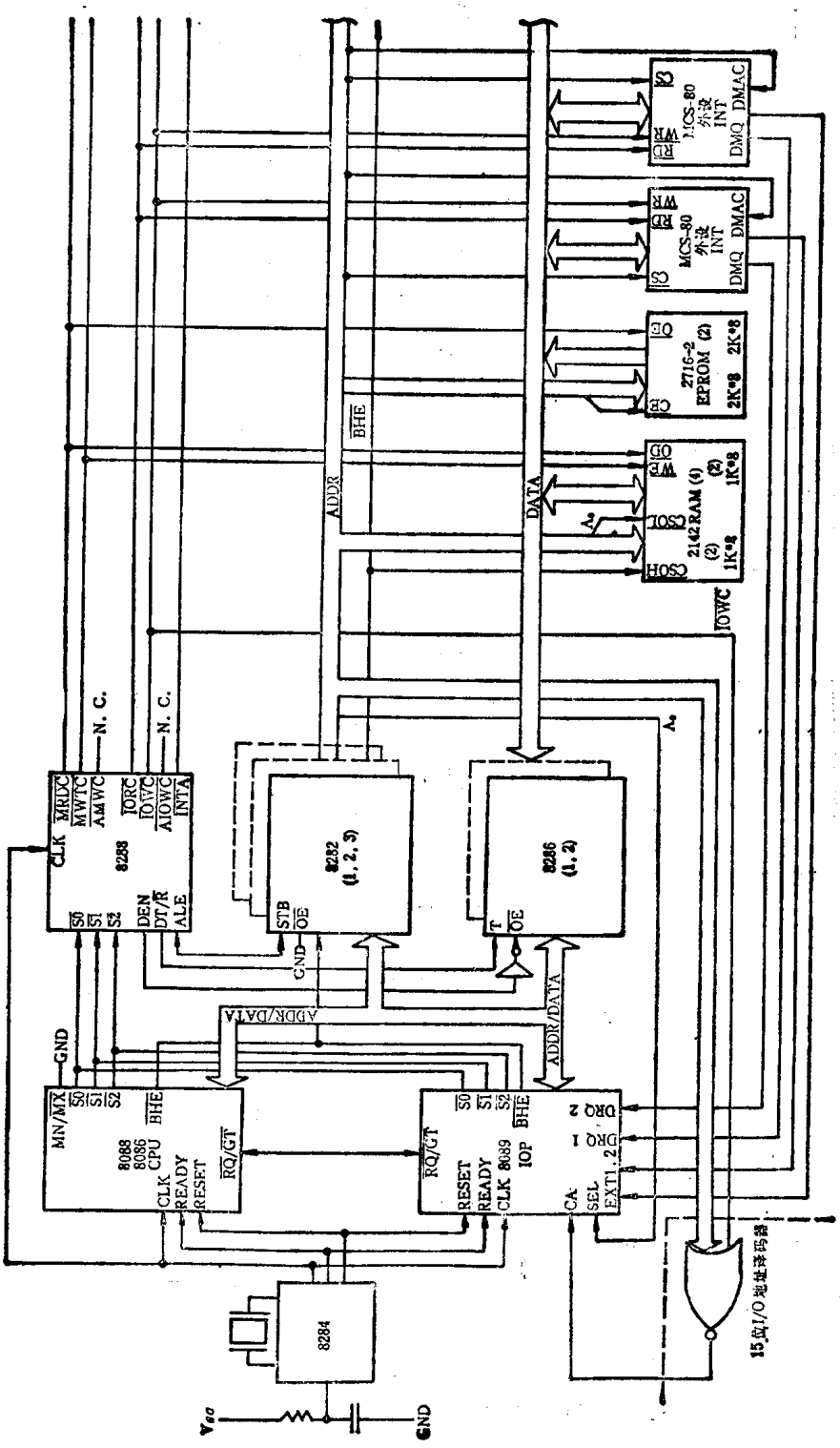


图 9.7　本地方式的 iAPX 86/11, 88/11 结构图

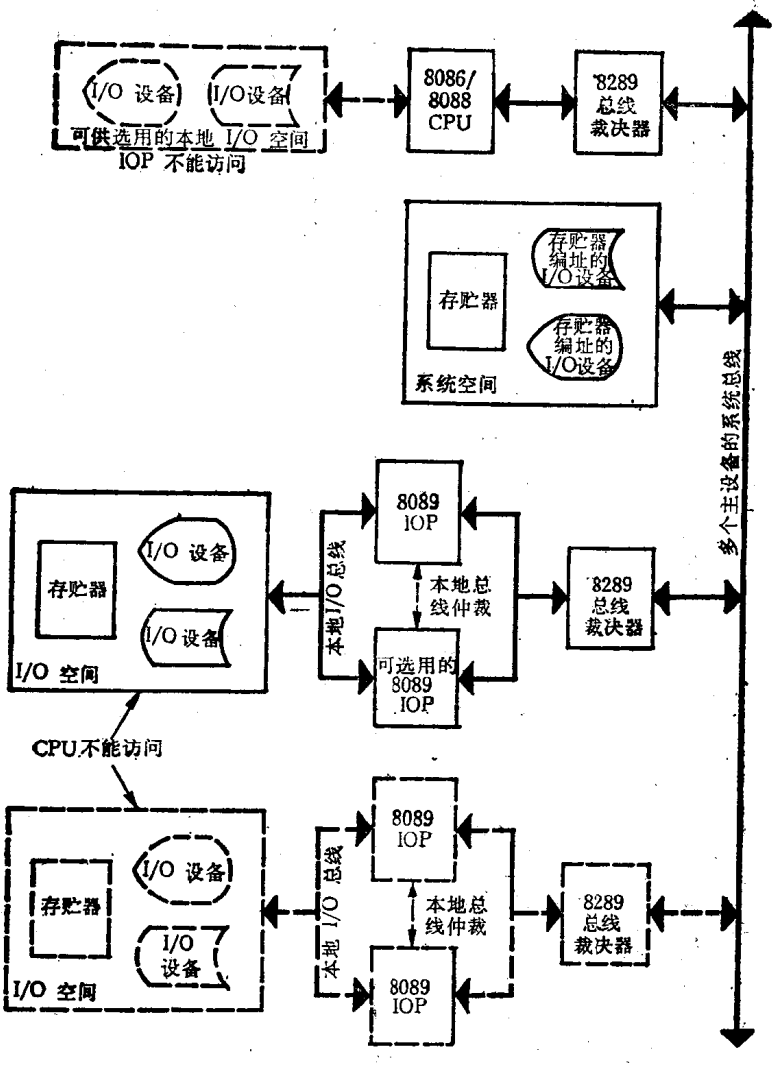


图 9.8　远程方式结构示意图

问这些模块。正是因为 8089 的 I/O 总线与 CPU 的系统总线分开了的缘故，8089 和 CPU 之间并行操作的能力(程度)才得到了大大的提高，系统的吞吐能力得到了增强。为此所付出的硬件代价是增加了系统总线裁决器和系统总线与本地总线之间隔离用的缓冲器。图 9.8 还给出了两个 8089 可以共享本地 I/O 总线的情况。下图给出了 8089 与 CPU 共享本地总线的一种结构，在该系统当中，8089 与其中的一个 CPU 组成本地方式结构，而与另一 CPU 组成远程方式结构。这种结构对于计算量大而输入/输出量相对较小的应用是较适宜的。

图 9.10 更具体地表示出 8089 的远程方式结构。从图中可以看出，8089 的总线与系统总线之间在物理上是分离的，其分离是通过使用总线收发器/锁存器实现的。8089 保持自己的本地总线，并且能对局部存贮器和系统存贮器实现访问。8089 与系统总线之间的接口包括以下几个部分：

1) 最多需要三个 8282 缓冲器/锁存器，用来锁存系统总线的地址。

2) 最多可要两个 8286 八位总线收发器，以实现系统数据总线的缓冲。

3) 一个 8288 总线控制器，利用它产生系统总线和本地总线的读/写控制信号，以及控制双向数据驱动器(8286)和地址锁存器(8282)的信号。

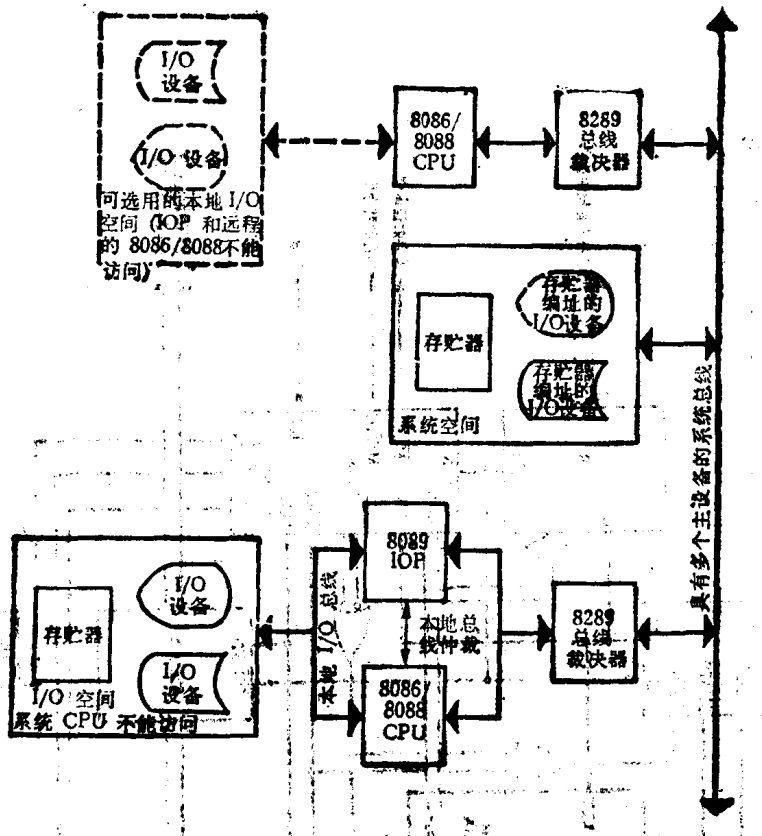


图 9.9 具有本地 CPU 的 8089 远程结构

4) 一个 8289 总线裁决器，分配系统总线使用权所需要的所有操作由它来完成。用它来代替本地方式中的请求/批准 (RQ/GT) 机构。该裁决器对 8089 状态线上周期信息的类型进行译码，从而确定 8089 是否有必要在系统总线上进行一次传输过程。

从图中可以看出，外部设备全部接在它们自己的数据总线和地址总线上，8089 与外设之间的通信不影响系统总线的操作。根据负载情况，也可以在局部总线上使用一些缓冲器/锁存器。如图左侧的 8286/87。为了进一步少用系统总线，还可以将 I/O 程序放在本地存贮器当中，即在局部(本地)总线上可以挂上 RAM 存贮器。

在这种远程方式中，8089 的本地总线只能访问 64K 字节的 I/O 空间，但其系统总线仍然可以访问 1 兆字节的存贮器空间。8288 总线控制器发出的 I/O 命令用于控制 I/O 总线，而它的存贮器读/写命令则用于控制系统总线。8289 总线裁决器与 8288 总线控制器通过对 8089 的状态线 $\overline{S2}$——$\overline{S0}$ 的译码，产生出访问系统总线或本地 I/O 总线的控制信号。

从上面介绍的两种结构方式中，我们可以看到，通道地址是通过地址线(A_0～A_{15})和 I/O 写命令($\overline{IOWC}$)确定的，即 A_1～A_{15} 与 $\overline{IOWC}$ 译码产生 CA (通道注意)信号，A_0 确定 SEL 的状态。外部设备的请求通信与中断信号(结束信号)分别接到 8089 的 DRQ 和 EXT 输入端，以实现同步通信。

两种基本结构方式的区别就在于对总线的共享在程度上有所不同。但 8089 的总线结构总是与工作在最大模式下的 8086/8088 的总线结构相似的，只要发出使用总线的请求，例如在执行通道程序期间取指令或进行数据传送时，就执行一个总线周期。8089 的总线周期和 8086/8088 一样也是由四个 T 状态组成的，也采用了地址/数据总线的分时复用技术。从下面的读/

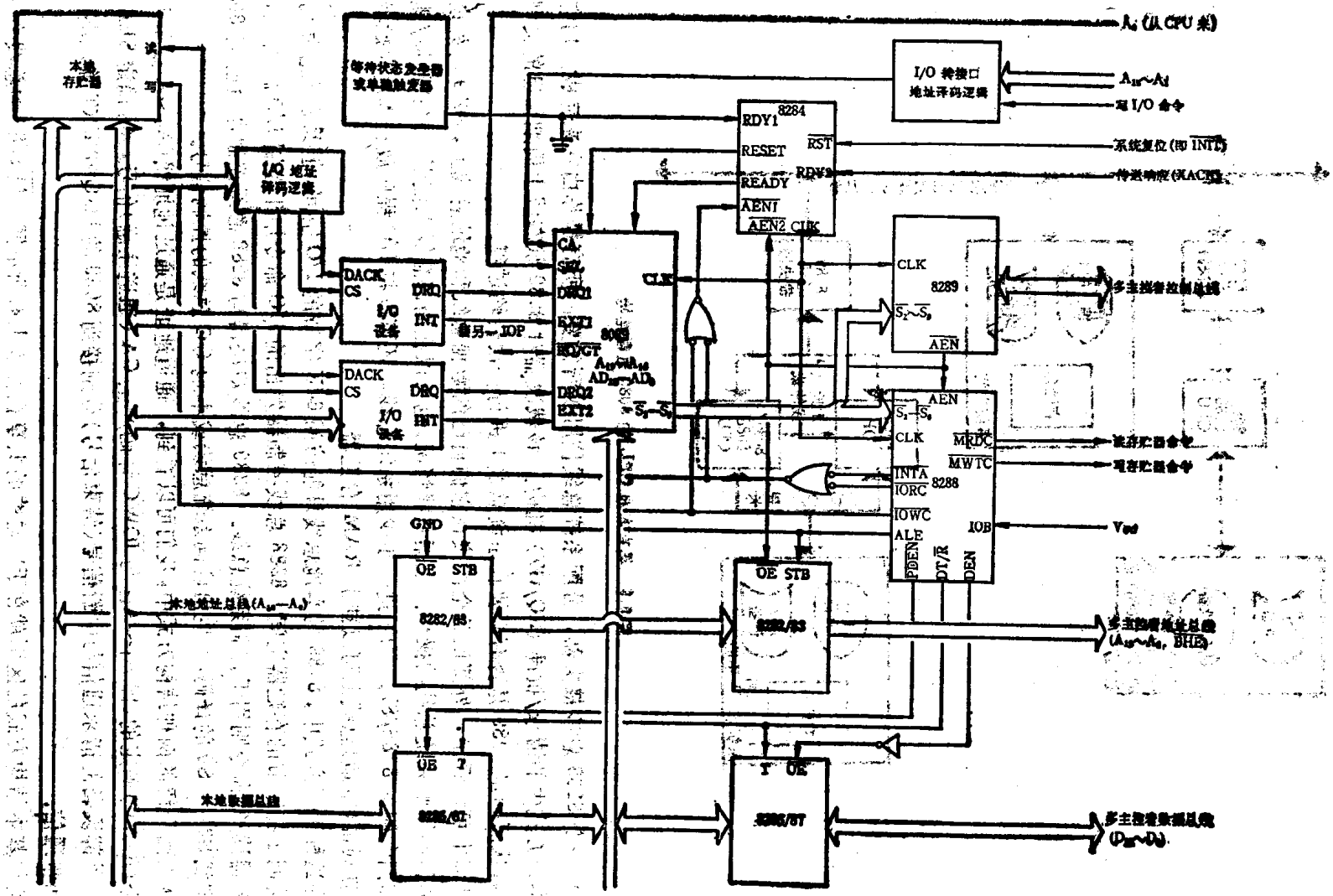
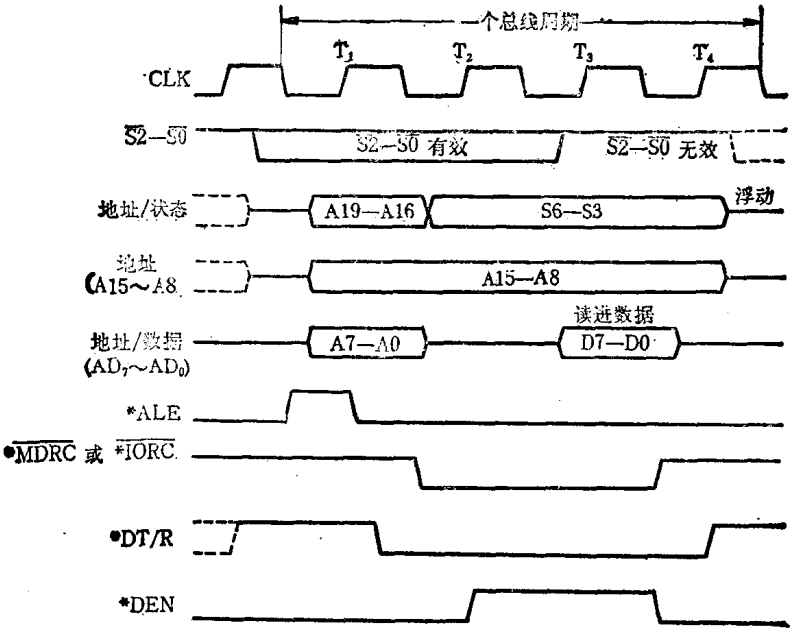


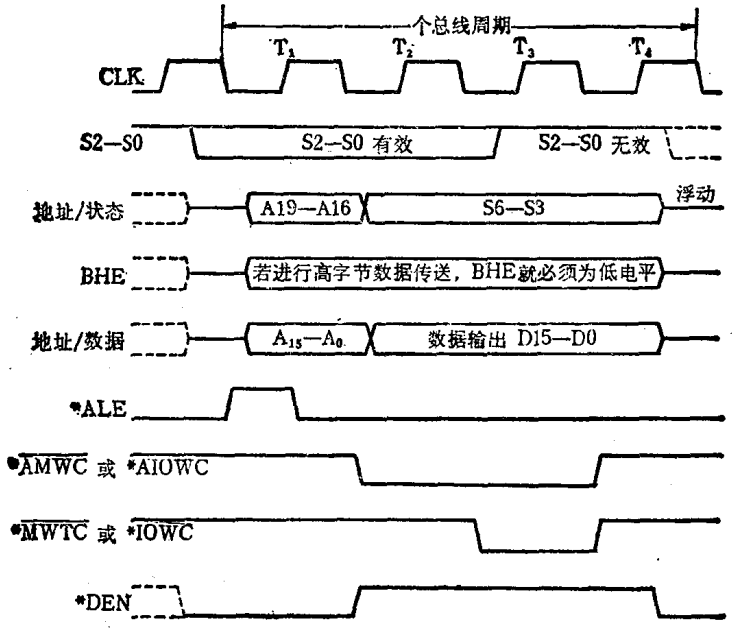
图9.10 典型的8(8)远程方式结构

写时序图中可以看出，地址总是在状态 T_1 中输出，并在 T_1 中发出 ALE 信号，用其后沿锁存地址信息；在状态 T_2 当中，对于读周期来讲(图 9.11)，地址/数据线就处于浮动状态，若为写周期(图 9.12)，就把所需写的数据送到这些总线上，在 T_3 状态期间，若为写周期，欲写的数据仍然保持在总线上面，若为读周期，也应读出数据；总线周期结束于 T_4 状态。



注：带*者为 8288 总线控制器的输出信号

图 9.11 读总线周期(8 位总线)



注：带*者为 8288 总线控制器的输出信号

图 9.12 写总线周期(16 位总线)

由于 8089 既能用 8 位总线、又可用 16 位总线来传送数据，因而在初始化期间应该设置好总线的宽度。当规定采用 8 位数据总线时，AD_{15}～AD_8 在整个总线周期内都表示地址，并在一个周期内维持不变，如图 9.11 所示，在这种情况下，若不要增大驱动能力，就不再需要外加地址锁存器了。从图 9.11 中，我们也可以清楚地看到，8 位数据总线与 8088 CPU 以及 MCS-85 中的许多外部设备（如 8155、8185 等芯片）是相容的。

状态线 $\overline{S_2}$～$\overline{S_0}$ 指明应执行的是何种总线周期，8288 总线控制器利用这三根状态线来产生所有存贮器和 I/O 的读/写控制信号，其编码如下表所示：

表 9.1 总线周期译码

状态输出			所表示的总线周期	8288 总线控制器控制输出信号
$\overline{S_2}$	$\overline{S_1}$	$\overline{S_0}$		
0	0	0	从 I/O 空间取指令	$\overline{INTA}$
0	0	1	从 I/O 空间读数据	$\overline{IORC}$
0	1	0	把数据写到 I/O 空间	$\overline{IOWC}$, $\overline{AIOWC}$
0	1	1	不用	无
1	0	0	从系统存储器取指令	$\overline{MRDC}$
1	0	1	从系统存储器读取数据	$\overline{MRDC}$
1	1	0	把数据写到系统存储器	$\overline{MWTC}$, $\overline{AMWC}$
1	1	1	无作用（无源）	无

状态线 S_6～S_3 是与最高位地址线 A_{19}～A_{16} 分时复用的，在总线周期的 T_2 状态，S_6～S_3 才有效，S_5 和 S_6 能使用户在多处理器的环境中确定是哪个处理器正在执行总线操作，对 8089 来讲，这两根状态线为全"1"，因此 S_6S_5 为 11 时代表 8089，这样就把 8089 与其它处理器区分开来了（如 8086 及 8088 中，S_6 总为"0"）。状态线 S_4 和 S_3 确定的是通道 1 或通道 2 所执行的总线周期的类型。编码情况见下表：

表 9.2 总线周期的类型编码

状态输出		周期类型
S_4	S_3	
0	0	通道 1 DMA 传送
0	1	通道 2 DMA 传送
1	0	通道 1 非 DMA 传送
1	1	通道 2 非 DMA 传送

每次 DMA 传送都至少需要两个总线周期，每个总线周期都不会短于 4 个时钟周期。当有关的存储器或 I/O 设备在规定的时间内不能响应，或在远程方式中 8089 必须等待到能使用系统总线时，8089 就在总线周期中插入等待状态，其插入规则也与 8086 及 8088 中的有关规定相同，即由 8284 时钟发生器来控制等待状态的插入，在需要插入时就将此等待状态插在 T_3 和 T_4 状态之间。

作为本节的结尾，也作为 8089 功能的一个大致的说明，我们来看一个例子，看看是如何利用 8089 来从软盘中读取记录信息的。当然，该例并没有说明 8089 的全部功能，而仅仅是概括了一下 8089 的基本操作情况以及它与 CPU 之间的相互关系。

为了访问软盘，CPU 首先必须获得对某个通道的占用权。这就要求检查该通道所对应的

控制块中的BUSY标志，只有在BUSY等于0时，才能占用该通道，并要求立即把BUSY标志置为OFFH，以防止其它任务或处理器获得该通道的使用权。CPU在获得所需的通道之后，就应填写参数块，此时，它应为通道提供这样一些参数：软盘控制器的地址、存放数据的缓存地址、驱动器号、及要读的信息位于软盘上的磁道号和扇区号，此外，参数块中还为8089返回操作结果保留了一些空间。参见下图：

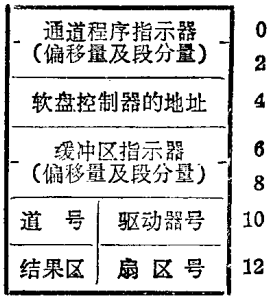


图9.13 参数块示例

在设置好了参数块之后，CPU就将"启动通道程序"命令写到该通道控制块的CCW字段当中，然后，CPU把所要求的通道程序的地址送至参数块，并把参数块的地址送到控制块(CB)中，从而把参数块、任务块及控制块连了起来。应该注意，在本例中，我们假定CPU知道诸如读磁盘这些通道程序的地址，故可以将这些地址填进参数块，但更常用的办法是将功能码(读、写、删去等)置入参数块，从而使通道程序根据所要求的功能的不同去执行不同的子程序。类似的问题在上面介绍的参数块中也存在着，我们要求CPU填进参数块的是有关外部设备的一些具体信息，可是对于今天的大多数系统来讲，更流行的是将CPU与I/O设备的具体特征相"隔离"，而是用人们较容易接受的文件名、记录名等等"名字"来访问外部设备。在此之所以介绍较"低级"的系统，主要是考虑到这也许会使我们的读者更容易掌握8089的有关内容。

现在，CPU就可以发出通道注意信号来命令8089执行通道程序了。对于I/O映象的8089，通常用OUT指令来给该通道发通道注意信号，对于存贮器映象的8089，则可利用MOV指令或其它访问存贮器的指令来调度该通道。接下来，通道就开始执行其地址已由CPU置入参数块的通道程序，由通道程序来预置8271软盘控制器，亦由通道程序预置并控制DMA传送的通道寄存器。

在8271及该通道都准备好了之后，通道程序就执行一条XFER指令。由于8271在收到最后一个参数(扇区号)之后，立即进入DMA传送方式，因此把这最后一个参数放在XFER指令之后传送，一旦该参数传送完毕，就开始DMA传送过程了。即当8271向通道发出DMA请求时，就开始DMA传送，该传送过程直到8271发出中断请求为止，2871的中断请求信号表示数据已传送完毕或出现了传送错误，8271的中断请求线接到8089的通道的外部终止引脚上(若为通道1就应接到EXT1)，这也就是讲，通道把中断请求理解为外部终止条件。通道在接到传送结束信号之后，继续执行通道程序，读出8271结果寄存器中的内容，以确定是否成功地从软盘中读出了数据。倘若传送中出现过"软"故障(可修正的错误)，则8089就再进行一遍DMA传送过程；假设查出硬故障(不可修正的错误)，或无故障(说明传送是成功的)，则8089就将结果寄存器的内容送到参数块的结果字段上，以便CPU查寻。然后，通道中断CPU(即通知CPU它已经将数据传送的请求处理完毕)，最后进入暂停状态。

CPU在响应了该中断之后，对参数块的结果字段进行检查，从而证实缓冲区中的内容是否

正确，若正确，CPU 就可使用这些数据；否则，说明传送有错，CPU 往往执行一个出错的子程序。

图 9.14 给出了该处理过程的一个粗流程图。

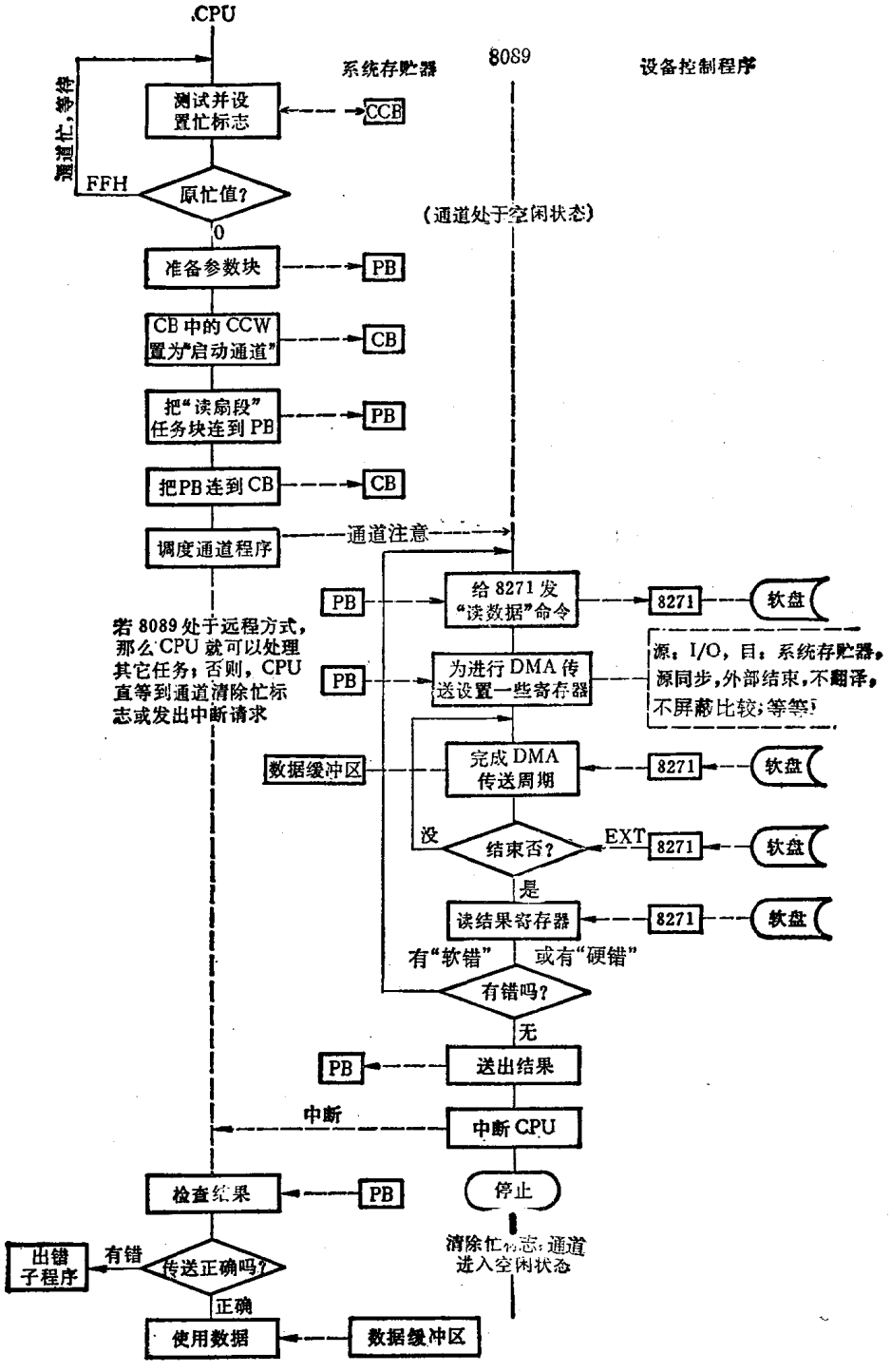


图 9.14　读软盘处理过程示意图

§9.3 8089的体系结构

§9.3.1 框图及引脚功能介绍

8089 共有 40 条引脚，它采用了双列直插式封装技术，它是由N沟道HMOS工艺制成的一个高性能的I/O处理器。图 9.15 和图 9.16 分别表示 8089 I/O 处理器的方框图和引脚结构，下面先来简要地介绍一下 8089 的引脚结构及其功用。

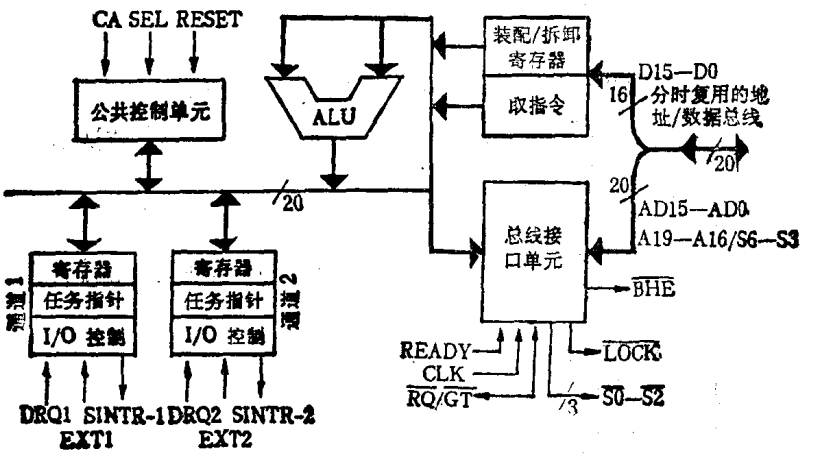


图 9.15 8089 I/O 处理器的方框图

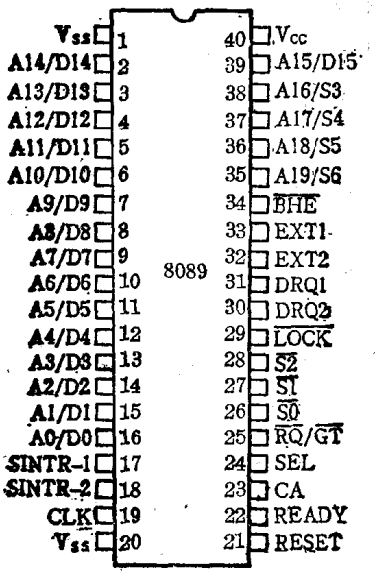


图 9.16 8089 的引脚结构

A0～A15/D0～D15，输入/输出

分时共享的地址/数据总线，其具体功能是由 $\overline{S0}$、$\overline{S1}$ 和 $\overline{S2}$ 这三根线的状态确定的。在把数据送到象 8088 这样的 8 位数据总线上时，A8～A15 是固定不变的，它总是代表地址。但在将数据传送到 16 位物理总线上时，A8～A15/D8～D15 也都是多路分时共享的。

A16～A19/S3～S6，输出信号

地址/状态线(Address and Status)，它们代表最高的4位地址线及状态信息，也是分时公用的。这4根引脚上的信号只有在存贮器寻址周期的T_1状态才代表地址，而在其它情况下，都当状态使用。状态编码如下：

S6	S5	S4	S3	
1	1	0	0	通道1上的DMA周期
1	1	0	1	通道2上的DMA周期
1	1	1	0	通道1非DMA周期
1	1	1	1	通道2非DMA周期

$\overline{BHE}$，输出信号

高8位总线可用控制线(Bus High Enable)，利用它，能对数据总线上的高8位(D8～D15)数据进行处理。只有当$\overline{BHE}$为低电平时，才可以把某个数据字节送到数据总线的高8位上。

$\overline{S0}$、$\overline{S1}$、$\overline{S2}$，输出信号

状态(Status)引脚，它们定义了8089在任一给定周期内的活动。编码如下：

$\overline{S2}$	$\overline{S1}$	$\overline{S0}$	
0	0	0	从I/O空间取指令
0	0	1	从I/O空间取数据
0	1	0	把数据写到I/O空间
0	1	1	未用
1	0	0	从系统存贮器中取指令
1	0	1	从系统存贮器中取数据
1	1	0	把数据写到系统存贮器
1	1	1	无源(无作用)

总线控制器和总线裁决器利用这些状态线产生所有的存贮器和I/O控制信号。在T_3或T_W状态中返回到无源状态(111)表示一个周期的结束，若要进入一个新的总线周期，就得在T_4状态改变这些状态信号。

READY，输入信号

准备好(Ready)，来自被寻址设备的一个准备好输入信号表示该设备已经准备好进行数据传输。

$\overline{LOCK}$，输出信号

锁住(LOCK)信号被用来通知总线控制器，需要使用总线不止一个连续周期。该信号是在TSL指令执行期间，通过通道控制寄存器设置的。

RESET，输入信号

总清(Reset)信号使得8089挂起它的所有活动，并且进入空闲状态，直到收到一个通道注意(CA)信号为止。这个信号必须保持4个时钟周期以上的高电平才算有效。

CLK，输入

时钟(Clock)为8089的内部操作提供了需要的所有定时信号。

CA，输入信号

通道注意(Channel Attention)信号引起8089对通道的注意。在该信号的下降沿，通过设置SEL就可以确定8089是主设备还是从设备(Master/Slave)或是确定所选择的是通道1还是通道2(CH1/CH2)。

SEL：输入信号

选择(Select)信号，在系统总清之后收到的第一个CA信号可以通过SEL告诉8089它是主设备还是从设备(SEL为0时，表示8089是以主设备的身份出现的，为1时，则表示8089为从设备)，与此同时，开始8089的初始化进程。对于任何其它的CA信号，SEL所解决的是选择哪个通道的问题(SEL为0时选择通道1，为1时选择通道2)。

$\overline{RQ}$/GT：输入/输出

请求/批准(Request Grant)信号，在本地方式中，为了在8089与CPU中间选取一个，让其占用系统总线，或在远程方式结构中，为了确定由哪个8089占用总线，都需要使用该请求/批准引脚，作为通讯会话的工具。

DRQ1—2：输入信号

数据请求(Data Request)，这是两个DMA请求输入端，该信号通知8089有一个外部设备已准备好发送/接收数据，它们所要使用的通道分别为通道1和通道2。在开始适当的存取操作时，才可以将此信号置为高电平。

$\overline{INTR}$1—2：输出信号

中断信号(Signal Interrupt)，SINTR1及SINTR2分别由通道1和通道2发出。这种中断信号可以直接送到CPU，也可以通过8259 A中断控制器送给CPU。利用它们可以通知系统由用户定义的事件已经发生。

EXT1—2：输入信号

这是两个外部终止(External Terminate)信号，它们分别送给通道1和通道2。倘若通道控制寄存器把外部信号规定为DMA传送的结束条件的话，EXT信号将引起当前DMA传送的结束。此信号必须保持高电平直至完成结束动作。

Vcc：输入、+5伏电源

Vss：接地线

我们也可以把8089看成是由多个功能块组成的，在图9.16当中，实际上已经将8089划分成了六个功能单元，它们是：通用控制部件、算术逻辑部件、装配/拆卸寄存器、取指令单元、总线接口部件和通道。各功能部件之间是用20位的数据总线连接起来的，这也有利于提高内部信息的传输速率。下面就分别对8089的各功能部件进行讨论。

§9.3.2 公用控制部件

公用控制部件(CCU)的主要作用就在于协调8089中各个功能部件的操作。它在8089中起着一个控制、协调中心的作用。

在8089当中，8089的所有操作，包括取指令、DMA传送、通道注意命令的响应等操作，都是由一些更为基本的活动即所谓的内部周期所组成的。一个总线周期可以用一个内部周期来完成，执行一条指令则有可能需要多个内部周期。8089中共有23种不同类型的内部周期，每个内部周期的执行时间为2～8个时钟周期，这还不包括可能插入的等待状态和总线裁决所占用

的时间. 公用控制部件主要就是通过给各个功能部件分配内部周期来实现控制、协调功能的，它能决定哪个部件将执行下一个内部周期. 例如，假设两个通道都处于活动状态，即都可以工作，公用控制部件可以确定哪个通道的优先级别高，然后就让优先级高的通道执行下一个内部周期；若两个通道的优先级别相同，那么就让两个通道交替运行. 此外，公用控制部件还可以对处理器 8089 进行初始化处理.

§9.3.3 算术逻辑部件

算术逻辑部件(ALU)类似于 CPU 中的 ALU，它也是专为完成某些算术逻辑运算而提供的，只是它的功能较弱，只能完成一些较简单的运算而已，这也是由它作为输入输出处理器这一特点所决定的，因为一般来讲，对输入/输出数据的处理要求都是比较低的. 在 8089 当中，ALU 能够进行不带符号的 8 位或 16 位 2 进制数的算术运算，算术运算的结果最长可达 20 位. 算术运算指令包括加法、加 1 和减 1，逻辑运算包括对 8 位或 16 位量的逻辑"与"、"或"、"非"运算.

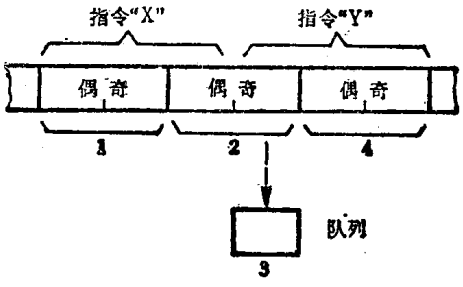
§9.3.4 装配/拆卸寄存器

所有进入 8089 的数据都要经过装配/拆卸寄存器. 前面已经介绍过，利用 8089，使得数据总线宽度不同的两个设备之间的通信成为可能，而利用装配/拆卸寄存器又使这种通信能够在最少的总线周期之内完成. 当在不同宽度的总线之间进行数据传送时，为了传送一个字，8089 需要三个总线周期，而不是需要四个总线周期，从而优化了传送过程. 当将一个 8 位总线映射到一个 16 位总线时(源的数据总线为 8 位，而目标为 16 位)，需要执行两次读操作以读一个字(装配)；反之，即当将一个 16 位总线映射到一个 8 位总线时，则需不止一次地执行写操作，即读一次(读了一个字)，写两次(拆卸). 例如，在用 DMA 方式把数据从 8 位的外设送给 16 位的存贮器时，8089 先执行两个总线周期，取得两个 8 位数据，在装配/拆卸寄存器中装配成一个 16 位的数据，然后再用一个总线周期把 16 位数据送入存贮器. 在第一个总线周期和最末一个总线周期，若遇到的是以奇地址开头的字，8089 可自动地对此进行处理，以使整个传送的效率达到最高.

§9.3.5 取指令单元

取指令单元负责为正在运行的通道取指令. 如果取指令采用的是 8 位数据总线，那么每个总线周期只能取到一个指令字节；如果取指令采用的是 16 位数据总线，则每个总线周期可以取到两个字节的指令码. 为了减少取指令所费的总线周期数，取指令单元能够根据情况，使用一个 1 字节的排队机构. 每个通道都有它自己的排队机构，同一个 8089 中的两个通道的排队机构之间没有影响. 设置这种排队机构的理由是：一条指令所占有的字节数目可能不是偶数，而对于 16 位的取指数据总线来讲，它每次取来的是一个字(两个字节)，这两个字节中很有可能包括了下一条指令的头一个字节(如图 9.17 所示)，若情况果真如此，取指令部件就将自动地把下条指令的这头一个字节存放到该字节队列中. 当通道执行到下一条指令时，它就先

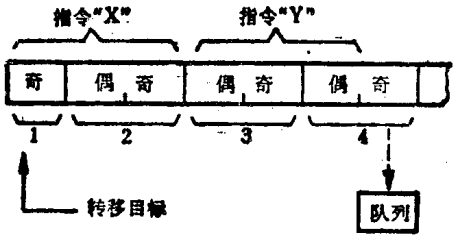
从此队列中取出该指令字节，而不再需要占用总线周期以取得该指令字节，因为通道从队列中取出的这个字节不需要占用总线周期，这是取指令部件节省取指总线周期的一种方法。



顺　序	指　令　字　节
1	"X"的头两个字节
2	"X"的第三字节加上存在队列中的"Y"的第一字节
3	从队列中取出的第一字节不占用总线周期
4	"Y"的最后两个字节

图 9.17　取指顺序(从偶址开始，16 位总线)

当遇到一条跳转指令或调用指令时，情况就有些特殊了，若转移的目标是一个奇地址，在用 16 位数据总线取指令时，取指部件就将用一个总线周期去取得该指令的第一个字节，如图 9.18 所示。这是因为程序已经改变了原来的顺序，从而使原来排队机构中的内容无效了的缘故。



顺　序	指　令　字　节
1	"X"的第一字节，奇地址，8 位总线周期
2	"X"的第二和第三字节
3	"Y"的第一和第二字节
4	"Y"的第三字节加上下一条指令的第一字节(它被保存在队列中)

图 9.18　从奇地址开始的取指顺序(16 位总线)

§9.3.6　总线接口部件

为了在 8089 和存贮器(或外部设备)之间传送指令或数据，需要这样的一个总线接口部件(BIU)，让它来执行所有的总线周期。总线的使用与寄存器的标志位有关，该标志位指出由 BIU 访问的是系统存贮器空间还是 I/O 空间。BIU 在状态线 $\overline{S_0}$、$\overline{S_1}$ 和 $\overline{S_2}$ 上输出总线周期的类型信息(例如从 I/O 空间取指令，把数据存入系统存贮器空间等)，总线控制器 8288 通过对状态 $\overline{S_0}$～$\overline{S_2}$ 译码，产生总线控制信号。

BIU 还把系统总线和 I/O 总线的实际宽度与逻辑宽度区分开来。无论是系统总线，还是 I/O 总线，它们的实际宽度都是固定不变的，并且在 8089 的初始化过程中都将各自的实际宽度告诉了总线接口部件。在本地方式中，系统总线与 I/O 总线的宽度必须相等；在远程方式的结构当中，8089 的系统总线的宽度必须与 CPU 系统总线的宽度相同，但由于 8089 的 I/O 总线是局部于 8089 的，故可对其实际宽度独立地进行选择，其宽度既可与系统总线的宽度相同，也可不同于系统总线的宽度。然而，若在 I/O 空间中存在 16 位的外围设备(或外围芯片)，就应该采用 16 位的 I/O 总线；若在 I/O 空间中只有 8 位的外围设备，则可以选用 8 位的 I/O 总线，如果考虑到便于与将来的 16 位芯片相组合，就应该采用 16 位的 I/O 总线结构。采用 16

位I/O总线的另一个优点就是，在通道程序存放在I/O空间中时，取指令的次数可以少一些。对于给定的DMA传送，通道程序规定了系统总线和I/O总线的逻辑宽度，每个通道都可以单独规定各自的逻辑总线的宽度。8位物理总线的逻辑宽度只能取8位，但是，对于16位的物理总线来讲，它既可以用作8位的也可以用作16位的逻辑总线。因此，用16位的物理总线就既能访问8位的外围芯片，也能访问16位的外围设备。表9.3列出了8089在本地方式和远程方式中物理总线(实际总线)宽度和逻辑总线宽度的组合情况。实际上，逻辑总线的宽度只是在DMA传送时才起作用，至于是以字节还是以字为单位取指令及读写数据则取决于总线的实际宽度。

表9.3 总线的实际宽度与逻辑宽度组合

结构方式	系统总线 实际的：逻辑的	I/O总线 实际的：逻辑的
本地	8:8	8:8
	16:8/16	16:8/16
远程	8:8	8:8
	16:8/16	16:8/16
	16:8/16	8:8
	8:8	16:8/16

注：/表示"或者"的意思

总线接口部件BIU除了负责传送指令或数据之外，还负责对本地总线进行管理。在本地方式结构中，BIU利用请求/批准线($\overline{RQ}/\overline{GT}$)从CPU取得总线控制权，也利用它在传送结束之后把总线控制权归还给CPU。在远程结构中，若存在一个本地的CPU(见图9.9)或另一个8089(如图9.8所示)，BIU也用$\overline{RQ}/\overline{GT}$来协调对本地I/O总线的使用。在远程方式的结构中，8289总线裁决器负责对系统总线的仲裁。BIU在执行指令TSL(检测并加锁)时，将自动插入总线封锁信号$\overline{LOCK}$，以便在DMA传送过程中防止别的处理器占用系统总线，从而保证传送过程不受其它处理器对总线的干扰。

9.3.7 通 道

虽然8089是单个处理器，但在大多数应用当中，我们都把它当作两个相互独立的通道来使用。通道既可以执行通道程序，也可以进行DMA传送，当然也可以处于空闲状态。下面着重介绍支持这些通道操作的硬件资源。

一、I/O控制部件

每个通道都有它自己的I/O控制部件，在DMA传送期间，它负责对通道的操作进行管理。如果采用的是同步传送方式，则通道在执行下次传送操作之前，就要等待DMA请求线上DRQ信号的到来，即只有在通道收到DRQ信号后，它才开始下次数据传送过程。若DMA传送是由外部信号来结束的，则通道就将不断地注视EXT(外部终止信号)输入端，并在该信号变为有效(高电平)时，停止DMA传送。在取周期到存周期期间，数据在8089当中，因而通道就可以有选择地对数据进行翻译、扫描、计数，还可以根据这些操作的结果来决定是否应该停

止传送．此外，每个通道都有一根 SINTR（系统中断）引线，通过该引线，通道可以利用软件的方法向 CPU 发出中断请求信号。

二、通道寄存器组

通道为了能够执行通道程序，实现 DMA 传送的功能需要一组寄存器。图 9.19 表示这样的通道寄存器组，为了使 8089 的两个通道能相互独立地工作，故给每个通道都设置了这样的一组寄存器。这些寄存器在执行通道程序时所起的作用与它们在 DMA 传送过程中所起的作用往往是不相同的，这一点从表 9.4 中就可以看出来。因此，如果在 DMA 传送之后，还要用到前面在执行通道程序时这些寄存器中的内容，则在 DMA 传送之前，通道程序应该把这些寄存器中的内容保存到存贮器当中。

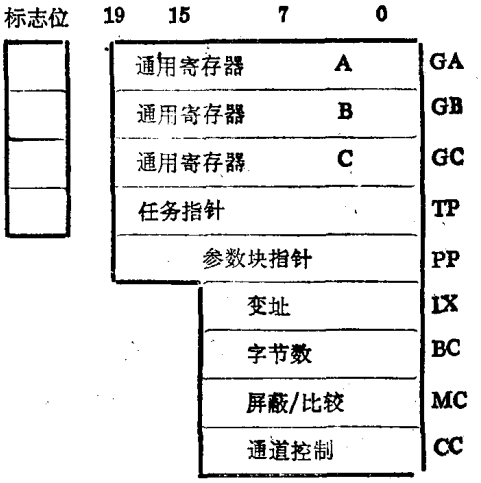


图 9.19　通道寄存器集

表 9.4　通道寄存器的用途

寄存器名称	位数	能否用程序修改	系统或 I/O 指针	在通道程序中的用途	在 DMA 传送中的用途
GA	20	能	可以	指令操作数	源/目标指针
GB	20	能	可以	指令操作数	源/目标指针
GC	20	能	可以	指令操作数	翻译表指针
TP	20	能	可以	程序指针	指示结束
PP	20	不能	系统	基址	无用
IX	16	能	无用	自动增量	无用
BC	16	能	无用	通用寄存器	字节数
MC	16	能	无用	屏蔽比较	屏蔽比较
CC	16	能	无用		通道控制

1．通用寄存器 A(GA)：通道程序可以把 GA 用作为通用寄存器或基址寄存器，当作为通用寄存器使用时，GA 存放 8089 指令的操作数；在被用作基址寄存器时，表示把 GA 用来对存贮器中的操作数进行寻址。在 DMA 传送当中，GA 是指向源或目标的指针，因此，在启动 DMA 传送之前，通道程序应该设置 GA 的内容，以让其指向 DMA 传送的源地址或目标地址。

2．通用寄存器 B(GB)：GB 的功用与 GA 相似，两者可以互相替换。只是在 DMA 传送

当中，一个指向源地址，另一个指向目标地址。也就是说，若 GA 指向 DMA 传送的源地址，则 GB 就指向 DMA 传送的目标地址，反之亦然。

3．通用寄存器 C(GC)：在通道程序执行期间，GC 可用作为通用寄存器或基址寄存器。如果在 DMA 传送期间要对数据进行翻译，通道程序就应在开始 DMA 传送之前把翻译表的首地址装入 GC。但在 DMA 传送过程中，不可以修改 GC 的内容。

4．任务块指针(TP)：当公用控制部件 CCU 开始或恢复执行通道程序时，它就从参数块中取出任务块指针的内容，并装到 TP 当中。在通道程序的执行期间，通道会自动地修改 TP 的内容，以使其指向下一条要执行的指令，显然，TP 相当于一般的指令指示器或程序计数器。程序跳转指令(JMP、CALL 等)也可以改变 TP 的内容，使程序不按顺序执行。当过程或子程序出现时，还要注意在调用时保护 TP 的内容，返回时将断点地址装入 TP。当然，也可以把 TP 用作通用寄存器或基址寄存器，然而，我们并不提倡这样使用，因为这会使程序很难理解。

5．参数块指针(PP)：CCU 在开始执行通道程序之前，要把参数块的地址装入寄存器 PP，该寄存器的内容是作为访问参数块中数据时的基址使用的，通道程序不能修改该寄存器中的信息，在 DMA 传送期间，PP 是没有用处的。

6．变址寄存器(IX)：在通道程序执行期间，IX 可作为通用寄存器来使用，同时，它也可以用作变址寄存器，以寻址存贮器操作数，在此，操作数的地址一般是把 IX 的内容加到基址寄存器形成的。作为变址寄存器使用时，在数组或字符串操作中，IX 能在指令的最后一步自动加 1。同样，在 DMA 传送过程中，IX 也没有用处。

7．字节数寄存器(BC)：在通道程序执行期间，BC 可用作通用寄存器。但其主要作用是在 DMA 传送过程中，假设要求在 DMA 传送了一定数目的字节数据之后停止 DMA 传输过程，则应在开始传送之前，将所要传送的字节数目装入 BC 寄存器。在 DMA 传送期间，不论是否把字节计数规定为传送的结束条件，每传送一个字节之后，BC 都自动减 1。但如果通道程序设定了在传送某个规定数目的字节数据之后停止传送的话，在 BC 等于 0 时，就结束当前的 DMA 传送过程。否则，即在没有把字节数设定为终止条件的情况下，即使 BC 的内容为 0，它仍将减 1，变成 FFFFH，并继续执行下去。

8．屏蔽/比较寄存器 (MC)：通道程序可以把 MC 寄存器用作为通用寄存器。但 MC 更多地是在通道程序或 DMA 传送过程中用于进行屏蔽比较，屏蔽比较的原理如图 9.20 所示，待比较的字节数值放在 MC 的低字节当中，屏蔽字节值位于 MC 寄存器的高字节。屏蔽字中的"1"用于选择待比较的字节数值中相应的位，"0"表示屏蔽掉(不管)比较值中对应的位。所以，图中高字节中的低 5 位为全"1"，表示对低字节中的低 5 位进行比较，即取出低 5 位的内容，从而得到 00100；屏蔽字节中的高 3 位全是"0"，表示屏蔽掉低字节的高 3 位，故进行比较

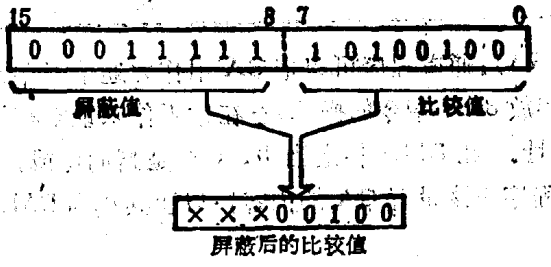


图 9.20　屏蔽/比较方法示意图

时，无需考虑高 3 位的内容。

9．通道控制寄存器(CC)：通道控制寄存器的内容用于控制 DMA 传送，其格式如图 9.21 所示。在开始传送操作之前，通道程序应把适当的数值装到该寄存器当中，关于 CC 的详细说明将在后面给出。CC 的第 9 位(加链位)是为执行通道程序服务的，它不属于 DMA 传送的范畴。若该位为 0，通道就以正常的优先级别运行；该位为 1，则将把通道程序的优先级提高到与 DMA 的优先级别相同，后面还将对优先级别问题进行讨论。此外，通道程序也可以把 CC 用作通用寄存器，但这容易影响加链位，即影响通道程序的优先级别，故我们不提倡这样来使用 CC 寄存器。

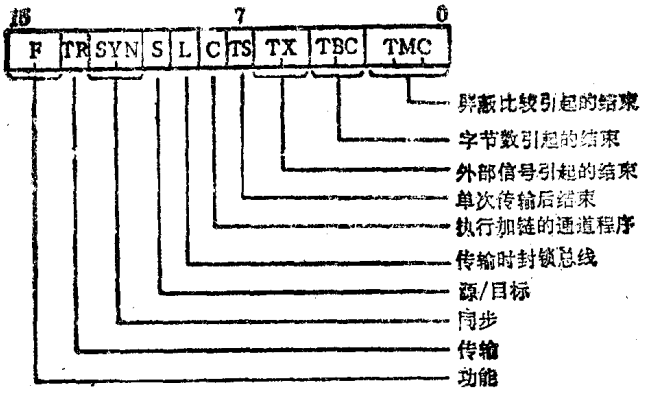


图 9.21　通道寄存器的格式

10．程序状态字(PSW)：每个通道都有自己的程序状态字，其格式如图 9.22 所示。利用 PSW 来记录通道的状态，设置 PSW 的目的就在于使通道的操作可以暂时挂起，在挂起一段时间之后能够继续执行挂起的操作。通道程序不能存取 PSW。当 CPU 发出“挂起”命令时，通道就把 PSW、任务块指针、任务块指针的标志位送到参数块的开头 4 个字节中，保护起来，如图 9.23 所示。在收到“恢复”命令时，就从参数块的保护区域中取出 PSW、TP 及 TP 的标志位，然后恢复通道的操作。

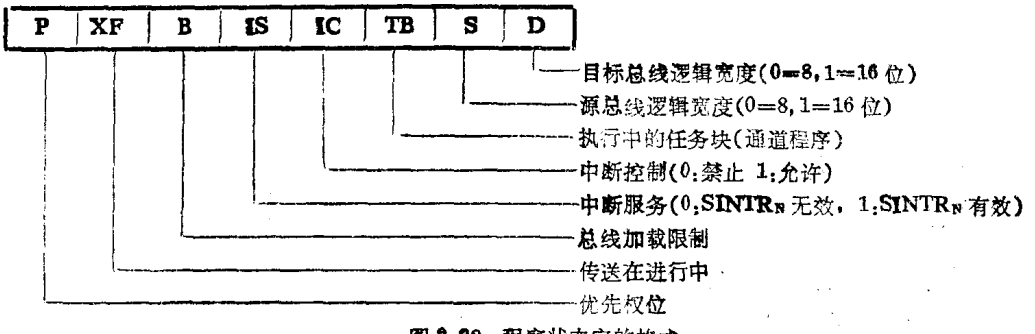


图 9.22　程序状态字的格式

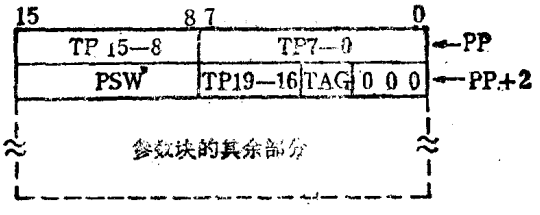


图 9.23　通道的状态保护区

11．标志位：寄存器 GA、GB、GC 和 TB 通称为指针寄存器，这是因为可把这些寄存器用于对系统空间或 I/O 空间进行寻址。由于 I/O 设备(芯片)也是按存贮器进行编址的(存贮器映象 I/O)，故这些指针寄存器就既可以寻址存贮器，也可以寻址 I/O 转接口。这些寄存器对应的标志位分别表示它们是指向系统空间(标志位为 0)，还是指向 I/O 空间(标志位 = 1)。若这些寄存器指向系统空间，它们就代表 20 位的地址，从而可对 1 兆字节的系统空间进行直接寻址；若指向 I/O 空间，则其低 16 位用作访问 64 K 字节的 I/O 空间的地址，而高 4 位不起作用。

若 CPU 发给 CCU 的命令是"启动系统空间的通道程序"，则 CCU 就将 TP 的标志位置 0；若命令为"启动 I/O 空间的通道程序"，CCU 就将把 TP 的标志位设置为 1。通道程序利用各种装入寄存器指令可以达到改变 GA、GB、GC 及 TP 的标志位的目的。一般来讲，"装入指示器"指令将把给定的指针标志位置为 0，表示装入的是一个系统地址；而"传送指针"指令则要根据所传送的物理地址的实际情况，确定相应的标志位中的内容。

12．通道的并行操作

两个通道可以并行运行，但实际上，在任何时刻都只有一个通道在进行实际的操作。在每个内部周期结束时，CCU 都要从它的两个通道中选出一个来执行下一个内部周期，在两个通道中的这种替换不产生附加的系统开销。8089 的优先权机构负责作出是否进行通道替换的决定，即由该优先权机构确定该由哪个通道投入运行。由于优先级别反映出了各种活动的相对重要性，当然应该让较重要的活动先进行，故 CCU 总是让优先级别较高的通道操作先投入运行。倘若两个通道的优先级别相同，CCU 就去查看 PSW 中的优先权位，并让优先权位高的那个通道先运行；若两个通道的优先权位相等，就让两个通道交替进行操作。通道所能执行的每种操作都具有一定的优先级别，这些优先级别大致反映出了各种操作的相对重要性。表 9.5 列出了各种通道操作的优先权级别以及一种操作被另一种操作所替换的边界条件。

表 9.5　各种通道活动的优先权和替换的边界条件

通道活动	优先权级别 1=最高	替换条件	
		用 DMA	用指令
DMA 传送	1	总线周期[1]	总线周期
DMA 结束程序	1	内部周期	无
通道程序(加链)	1	内部周期[2]	指令
通道注意程序	2	内部周期	无
通道程序(不加链)	3	内部周期[2]	指令
空闲	4	两个时钟周期	两个时钟周期

1. DMA 在 $\overline{LOCK}$ 有效时不能替换

2. TSL 指令除外

在表 9.5 的通道活动一栏中，又引进了两种我们在前面没介绍过的通道操作，一种叫 DMA 结束程序，另一种叫通道注意程序。当 DMA 传送结束时，通道需执行一小段通道程序，利用这段程序来调整 TP 的内容，使用户程序能从建立 DMA 传送时所规定的地方恢复执行。

与此类似，当通道收到通道注意(CA)信号时，它也要执行一段内部程序，该程序检查通道控制字 CCW 中的内容，并执行 CCW 中的命令。DMA 结束程序和通道注意程序都是用 8089 的指令写成的，且都存放在 ROM 当中。

从表 9.5 中还可以看到，通道程序具有两种优先级，加链的通道程序的优先级为 1，不加链者为 3。通道程序是否为加链则由通道控制寄存器中的加链位所决定，若加链位为 1，通道程序就按 1 级运行，即把通道程序链成与 DMA 具有同样的优先级；若加链位是 0，通道程序则按优先级 3 运行。由此可见，利用加链的方法可以提高关键的通道程序的优先级。

根据通道操作的类型，CCU 只能按照边界条件在某些交界点上进行两个通道的替换。举例来说，假设 8089 的两个通道为通道 A 和通道 B，通道 A 正在进行操作，并且除非 CCU 选择通道 B 投入运行，它将继续运行下去。CCU 是否进行替换(从通道 A 替换为通道 B)则取决于通道 B 是在执行 DMA 还是在执行指令。从表 9.5 中可以看出，若通道 A 正在进行 DMA 传送，则它可在任何总线之后被通道 B 的 DMA 所替换(当通道 A 再次运行时，就从挂起点处重新继续执行)。当通道 B 执行 DMA 时，通道 A 就可比较快地被通道 B 所替换，因为若两个通道都在执行通道程序，则替换条件就以指令为边界了，也就是说，通道 B 必须等待通道 A 执行完一条指令之后才能替换它，而一条指令都是由几个内部周期组成的，显然通道 B 的等待时间就比较长了。此外，通道 A 在执行 DMA 结束程序和通道注意程序中不能被通道 B 的指令所替换，而只能被通道 B 的 DMA 所替换，由于这两种内部程序都比较短，故不会使通道 B 等待太长时间。表 9.6 概述了通道替换的大致情况。

表 9.6 通道替换举例

通道 A(上一内部周期在工作)				通道 B			结果
动作	加链位	优先权位	加锁	动作	加链位	优先权位	
DMA 传送	×	×	无效	空闲	×	×	A 运行
DMA 传送	×	×	无效	通道注意	×	×	A 运行到当前传送周期的结束，然后 B 运行
通道程序	×	0	无效	通道程序	×	1	B 运行
通道程序	×	0	无效	通道程序	×	0	两通道交替运行
通道程序	1	×	无效	通道程序	0	×	A 运行
DMA 传送	×	1	无效	通道程序	1	1	A 运行一个总线周期之后，B 运行一个总线周期或内部周期
通道注意	×	×	无效	通道程序	1	×	若 A 已开始运行则 A 运行，否则 B 运行
DMA 传送	×	×	有效	通道注意	×	×	A 运行到 DMA 结束
通道程序(TSL 指令)	0	×	有效	DMA传送	×	×	A 执行完TSL指令，当LOCK 无效时，B 投入运行

§9.3.8 8089 的存贮器结构

如前所述，8089 既可以访问系统空间，也可以访问 I/O 空间。其中系统空间与 CPU 的存贮器空间一致，它最多可含 1 兆(1048576 个)字节；I/O 空间既可以与 CPU 的 I/O 空间一致，也可以是 8089 的专用空间，它最多可包含 64 K (65536 个)字节。系统空间的存贮部件可用 8288 总线控制器发出的读存贮器和写存贮器命令来访问，而 I/O 空间的存贮器可通过 8288 的 I/O 读写命令来访问。

从软件角度来看，8089 把这两种存贮空间都看成是可单独寻址的不分段的字节序列，如

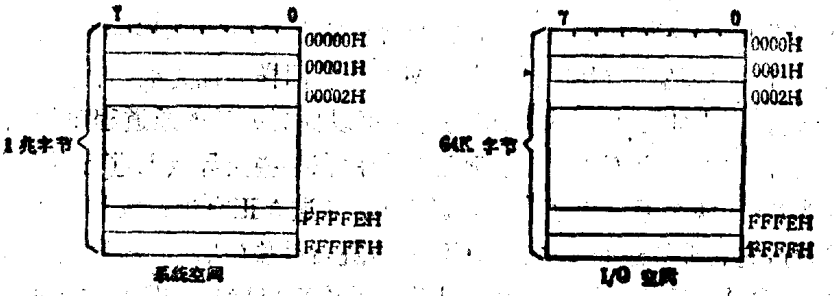


图 9.24 存贮器结构

图 9.24 所示。

尽管 8089 通常与 8086、8088 共享系统空间，但它们在对这一空间的处理上是不相同的。我们已经知道 8086 及 8088 中的存贮器单元有逻辑地址（16 位的段分量和偏移量）和物理地址(20 位)之分，但对于 8089 来讲，它看不见存贮器空间的分段式结构，它用 20 位的指针寄存器来访问系统空间中的所有物理单元，对 8089 I/O 空间的处理也与此类似，只是此时是用 16 位地址而已。

按照 Intel 的约定，字数据的高字节总是存放在较高的地址单元当中(见图 9.25)。8089 也能够识别 8086 和 8088 所用的双字指示器变量(见图 9.26)，该指示器的低地址字为偏移量，高地址字为段分量，当把双字指示器装到一个指针寄存器中时，8089 可将它变换成一个 20 位的物理地址。另外，还有一个称为物理地址指示器的专用 3 字节变量，专门用来保存和恢复各指针寄存器及其标志位，如图 9.27 所示。

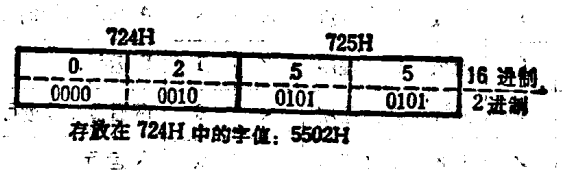


图 9.25 字变量的存贮格式

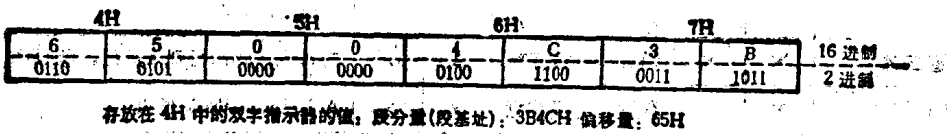


图 9.26 双字指示器变量的存贮格式

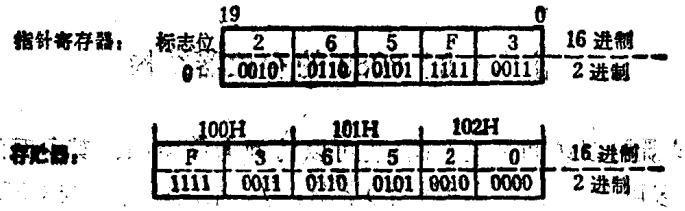
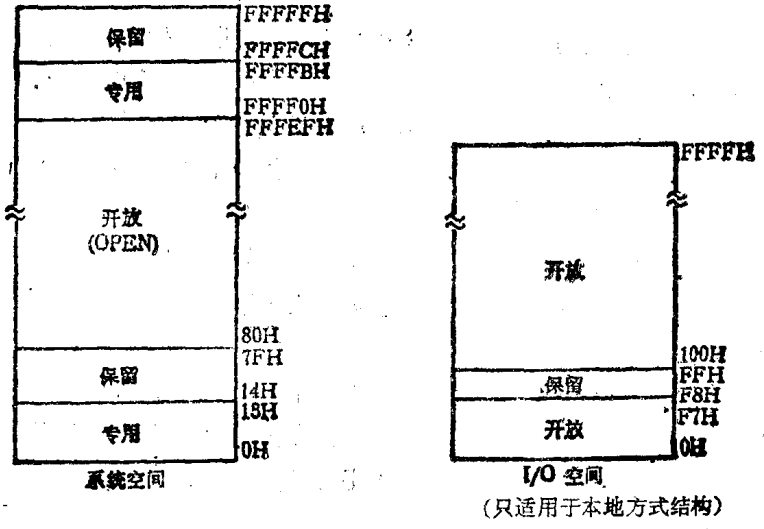


图 9.27 物理地址指针变量的存贮格式

为了获得与 Intel 软、硬件产品的兼容性，在存贮器中还专门保留了一些空间，这些单元是专用于处理器的某些功能或由 Intel 的其它硬件、软件产品所保留的。具体讲，单元 0H 到 7FH 这 128 个字节用作 8086/8088 的中断指示器，FFFF0H 到 FFFFFH 这 16 个字节用做 8086、

8088 及 8089 的启动程序，其它还有一些 Intel 的保留单元，这是系统空间的占用情况。对于 I/O 空间来讲，若 8089 处于本地工作方式，则其 I/O 空间与 CPU 的 I/O 空间一致，但必须注意它也有备用单元(F8H 到 FFH)；若为远程方式结构，则可不受任何限制地使用整个 I/O 空间。图 9.28 给出了存贮空间的组织情况。



(只适用于本地方式结构)

图 9.28 专用及保留的存贮器单元

8089 的程序与其在存贮器中的位置是无关的，其代码可在存贮器中"浮动"，但需要注意的是，整个通道程序必须作为一个整体来进行移动。这样做的优点是，通过在外存和内存之间进行程序传送，通过把分散的存贮器空闲单元(碎片)合并成更大的、更有用的连续空间，而使系统更为有效地利用有限的存贮器资源。另一方面，也应该看到，当 CPU 对程序浮动进行控制时，8089 又是实现这种通道程序、参数块及 CPU 程序实际移动的理想工具，因为用一个非常简单的通道程序以 DMA 方式传送代码，比用等价的 CPU 传送指令要有效得多，特别是在外存(磁盘)与存贮器之间传送时，效果就更为明显。

存贮器的访问总是利用一个指针寄存器及其标志位来实现的。标志位可以区分开所要访问的是系统空间(标志位 = 0)还是 I/O 空间(标志位 = 1)，指针寄存器中存放的是待访问单元的基址，即指针寄存器是作为基址寄存器使用的。若要寻址 I/O 空间，只要使用指针寄存器的低 16 位；而寻址系统空间则需用到整个的 20 位。各种类型的寻址所使用的基址寄存器情况请参见表 9.7。利用 8089 的寻址方式(后面将要介绍)就可以得到操作数的有效地址。

表 9.7 用于存贮器访问的基址寄存器

存贮器的访问类型	基址寄存器
取指令	TP
DMA 的源	GA 或 GB(由 CC 规定)
DMA 的目标	GA 或 GB(由 CC 规定)
DMA 翻译表	GC
存贮器操作数	GA 或 GB 或 GC 或 PP

从表 9.7 中可以看出，除了 GA、GB 和 GC 之外，还可以用寄存器 PP 来寻址存贮器操作数。PP 没有标志位，用它来寻址就意味着访问系统空间，事实上，参数块也确实总是驻留在

系统空间中的。还需注意，尽管通道程序可用 PP 来寻址参数块中的数据，然而它却不能读写 PP 寄存器。

8089 在初始化时，已经掌握了系统总线和 I/O 总线的实际宽度。若总线是 8 位宽的，则 8089 就象 8088 那样访问该总线上的存贮器，无论是取指令，还是读写操作数，每次存贮器访问(占用一个总线周期的时间)都只能取/存一个字节，存取一个字操作数需要两个总线周期，但这对于软件来讲是完全透明的。这时，8089 根据需要取出指令，而无需对指令流排队，其理由很简单，因为每次取一个字节不会把下一条指令的头一个字节取出来。

对于 16 位的总线来讲，8089 就象 8086 那样访问总线上的存贮器。当欲访问 16 位的数据时，若低 8 位恰好位于偶地址单元，高字节位于地址为奇数的单元，那么在一个总线周期内就可以访问一个字；反之，即若低字节是在奇数地址的存贮体中，则访问该字就需要两个总线周期，第一个总线周期用于在奇数地址上完成低字节部分的数据传送，第二个总线周期完成位于偶地址的高字节的传送。字节操作数总是以 8 位为单位来存取的。至于取指令的情况前面已经做过讨论，就不再重复了。

程序能否正确地工作与存贮器总线的宽度是完全无关的。当总线宽度增加到 16 位时，原来为使用 8 位总线的系统而编写的通道程序不需修改便可执行，在编写程序时，最好假定其运行环境为 16 位的总线，把字安排在偶地址中，因为这样一来，不管程序运行在何种系统中，它都能保证最有效地利用总线。

§9.3.9 输入/输出机构

8089 是 CPU 的编程 I/O 能力和 DMA 控制器的高速数据块传送特点相结合的产物，它比一般的 CPU 或 DMA 控制器有更大的灵活性。例如，它在 DMA 传送期间可进行比较和翻译等操作。8089 负责把数据从源地址传送到目标地址，至于这些地址所代表的单元是存贮器还是 I/O 则是无关紧要的，这是因为 8089 可以寻址所有的系统空间和 I/O 空间。此外，由 8089 控制的这种传送不仅可在某种空间之内进行，而且可在系统空间与 I/O 空间之间进行。

通道程序进行 I/O 处理所采用的方法类似于 CPU 与存贮器映象 I/O 设备的通信方式。为了完成数据传送，只要使用访问存贮器的指令，而不需要使用象 8086/8088 的 IN 和 OUT 那样的专用 I/O 指令，其理由就是 8089 不区分存贮器部件和 I/O 设备，这样一来，读/写一个存贮器字节或字操作数的任何指令，都可以用来访问 I/O 设备。这些指令从源操作数取得数据，对数据进行适当处理，然后把结果送到目标操作数。也就是说，当源操作数为一个 I/O 设备的地址时，指令就从该设备输入数据，当目标操作数引用某个 I/O 设备的地址时，就将数据从该设备输出。

大多数 I/O 设备控制器都带有一个或多个内部寄存器，利用这些寄存器来接收命令、提供设备的状态信息、给出操作的结果信息。对这些寄存器可进行如下处理：

1. 读、写整个寄存器；
2. 置位或清除寄存器的某些位，保持其余位不变；
3. 测试寄存器的某一位。

利用表 9.8 中列出的这些 8089 指令，可以比较有效地完成这些任务。

表 9.8 可用于 I/O 的访存指令举例

指　令	对 I/O 设 备 的 作 用
MOV/MOVB	读或写字/字节
AND/ANDB	清除字/字节的某些位
OR/ORB	设备寄存器中多位置"1"
CLR	清除(字节中的)某一位
SET	(字节中的)某位被置为"1"
JBT	若被测试位为"1"则转移
JNBT	若被测试位为"0"则跳转

由于存贮器访问指令常被用来完成编程的 I/O 工作，因而设备的寻址也就非常类似于存贮器的寻址。I/O 设备的基地址也从指定的指针寄存器中获得，它可以是 GA、GB 或 GC，也可以是 PP，但由于通常不把 I/O 设备置于参数块中，故一般情况下都不把 PP 用作 I/O 设备的基地址。然后再利用 8089 的某种寻址方式，产生相对于基址的偏移量，从而形成设备的实际地址。至于指针寄存器的标志位，它将设备定位于系统空间（标志位为 0）或 I/O 空间（为 1），若设备位于 I/O 空间，则仅用指针寄存器的低 16 位作为基地址；若定位于系统空间，则需用到全部的 20 位。

我们已经知道，8 位或 16 位设备都可以挂在 16 位的总线上面，但为了提高传送效率，使得有可能在一个总线周期之内完成一次字传送，我们应该设法把所有的 16 位设备置于偶地址。对于 16 位总线上的 8 位设备，其内部寄存器必须为相隔两个字节的全奇地址（如 1H，3H，5H 等）或全偶地址（如 2H，4H，6H，…），奇地址上的 8 位设备必须能在 16 位物理总线的高 8 位上进行数据传送。在用 16 位总线组成的系统中，利用字节指令访问所有 8 位外围设备，而用字指令访问 16 位设备。

挂在 8 位总线上的只能是 8 位设备，并且只能用字节指令进行访问。8 位总线上的 8 位设备可置于任何地址（奇地址或偶地址均可），且其内部寄存器是连续编址的（如 1H，2H，3H …），但是，若把它们指定为全奇地址或全偶地址，则有利于以后朝 16 位总线的转化。

对于 8089 的 I/O 总线传送，可用下表进行小结。其中的总线宽度是指物理总线的宽度，指令确定所要进行的传送是字节传送还是字传送，总线周期指的是为了完成该传送，8089 需要执行的总线周期数目。

表 9.9 编程 I/O 的总线传输

总线宽度	8				16			
指　令	字节		字*		字节		字	
设备地址	偶	奇	偶	奇	偶	奇	偶	奇
总线周期	1	1	2	2	1	1	1	2

* 通常不用

§9.4 DMA 传送

8089 除了能象 8086/8088 那样进行字节或字的编程输入输出之外，还能用 DMA 方式传送数据块。数据块的 DMA 传送可以在存贮器与存贮器之间、存贮器与 I/O 转接口之间、I/O

转接口与 I/O 转接口之间进行。倘若规定采用字节数作为结束传送的条件，则每个数据块不能大于 64 K 字节，在其它情况下，对数据块的大小没有限制。在进行 DMA 传送之前，通道程序必须发命令给外设控制器，并对传送中所要用到的寄存器进行初始化。在 DMA 传送期间不需要执行指令，但可以达到很高的传送速度。

§9.4.1 外设控制器的初始化

能进行 DMA 传送的外设控制器一般都是相当灵活的，它具有多种功能，能进行多种类型的操作。例如 8271 软盘控制器就能读一个扇段、写扇段、寻找 0 磁道等。这类控制器当中通常有一个或多个内部寄存器，在执行某种实际操作之前，一般得先对这些寄存器进行编程。

8089 的通道程序把这些设备寄存器看作是一些存贮器单元。通道程序在把设备（芯片）的基地址装入指针寄存器之后，就可以用编程的 I/O 指令与这些寄存器进行通信。对于某些控制器来讲，它们在接收到所有的控制器所需参数之后立即开始 DMA 传送。因此，在采用这种类型的控制器时，送出最后一个参数的通道程序指令应当紧跟在 8089 的 XFER 指令之后。

在这些内部寄存器当中，通常有一个用来存放状态信息的状态寄存器，通过它可以了解控制器是否正在工作、是否发生错误等有关信息。

§9.4.2 通道的准备工作

为了让某个通道执行 DMA 传送，必须为该通道提供一些与此次传送有关的操作信息，这一任务应由通道程序负责完成。通道程序把有关操作的信息装入通道寄存器，此外还要执行一条特殊的指令，以设置总线的逻辑宽度（见表 9.10）。

表 9.10 控制 DMA 传送的有关信息

信　息	寄存器或指令	必需的或选用的
源指针	GA 或 GB	必需的
目标指针	GA 或 GB	必需的
翻译表指针	GC	可选用
字节数	BC	可选用
屏蔽/比较数值	MC	可选用
逻辑总线宽度	WID	可选用（在 RESET 后必需执行一次）
通道控制	CC	必需的

一、源指针和目标指针

寄存器 GA 和 GB 用于存放 DMA 传送的源地址和目标地址，究竟哪一个是源指针，则要由通道控制寄存器的 S 位（第 10 位）所决定，但两者当中必有一个指向源，另一个指向目标。

源指针和目标指针既可以指向存贮器空间，也可以指向 I/O 空间，这是由 GA 和 GB 的标志位确定的。若它们指向存贮器单元，则应含有传送的开始地址，即缓冲器的最低地址，并且

在传送过程中，地址还要不断地加1，以便实现数据的成块传送；若标志位表明选择的是I/O空间，则指针寄存器的高4位就没有用处，故在I/O空间寻址的范围不会超过64 K字节。源和目标可以位于相同的地址空间，也可以位于不同的地址空间。

二、翻译表指针

如果在进行数据传送的同时，还要求对数据进行翻译的话，寄存器GC就应该指向一个由256个字节所组成的翻译表的首字节。翻译表可以放在系统空间或I/O空间当中，GC寄存器的内容及其标志位都可以用指令进行设置。翻译操作只能对字节数据进行，所以，源和目标的逻辑总线的宽度都必须是8位的。

通道对一字节数据进行翻译时，它把该数据当作一个不带符号的8位2进制数来处理，把这个数和GC的内容相加，得到一个存贮器单元的地址，即翻译表中某一单元的地址，再从该单元中取出相应的字节数据，这就得到了翻译之后的数据，最后把结果送回到原数据存放的单元中去。图9.29说明了这一翻译过程。

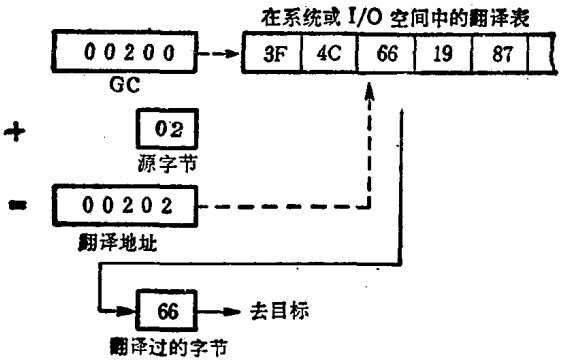


图9.29 数据的翻译过程

在上述的翻译过程中，GC是始终不变的，它总是保存着翻译表的始地址。若GC指向I/O空间，则其最高4位在操作当中不起作用。

三、字节数

如果要求DMA传送在传送了一定数目的字节数据之后结束的话，就应当事先把需传送的字节数作为一个无符号的16位数据存入寄存器BC之中。不论是否把字节数作为传送的结束条件，每传送一个字节，BC都将自动减1，只是若把它作为结束条件，在BC等于0时，就将终止当前的DMA传送。

四、屏蔽/比较数值

倘若规定传送的字节数据(也许是经翻译过的)与所查找的数值相等(或不相等)时结束传送，则应该按图9.20所示的那样来设置MC寄存器的内容。在传送过程中，MC的内容保持不变。在需要对传送的数据进行比较时，通常总是把目标总线的逻辑宽度设置为8位，当然这并不意味着16位的逻辑总线宽度不合法，只是在宽度为16位时，它只对低8位进行比较。

五、总线的逻辑宽度

8089的WID指令被用来为DMA传送建立源和目标总线的逻辑宽度。物理总线宽度为

8 位的任何总线，其逻辑宽度只能是 8 位的；但对 16 位宽度的实际总线来讲，其逻辑宽度可以是 8 位的，也可以是 16 位的。每个通道都可单独地设定自己的逻辑总线宽度。

在用 16 位的物理总线与 I/O 设备(芯片)进行数据传送时，应把逻辑总线的宽度设置成等于外部设备的总线宽度；在用 16 位的物理总线与存贮器进行通信时，为了获得最高的传送速度，则应把逻辑总线宽度设置为 16 位，当然也有例外，在下面这两种情况下，就必须把逻辑宽度设置成 8 位：

1．数据需经过翻译；

2．虽然传送的目标是 16 位的存贮器，但数据需要在屏蔽掉一些位的情况下进行比较。

利用一条 WID 指令，就可以同时设置源和目标总线的逻辑宽度，在设置好了之后，逻辑宽度保持不变，直到执行另一条 WID 指令或处理机复位时为止。在处理机刚刚复位时，其总线的逻辑宽度为一未定值，因此，在进行首次 DMA 传送之前，必须执行一条 WID 指令。

六、通道控制寄存器

16 位的通道控制寄存器 CC 被分成 10 个字段，这些字段规定了如何执行 DMA 传送(见图 9.30)。通道程序通过把一个字装入 CC 寄存器，就能达到设置这些字段之目的。下面分别对这些字段进行说明：

1．功能字段(第 15～14 位)用于规定源和目标是存贮器还是 I/O 转接口。在 DMA 传送的过程中，对存贮器和 I/O 转接口的处理是不一样的，这主要表现在：通道使指向存贮器的源或目标指针寄存器每传送一次自动加 1，以让数据能从连续的存贮空间中取出，或使数据在存贮器中能按地址的连续顺序存放。但是，对于指向 I/O 转接口(设备、芯片)的指针来讲，它在整个传送过程中保持不变。

2．翻译字段(第 13 位)规定数据是否需要进行翻译。若该字段为 1，就表示要求利用由寄存器 GC 所指向的翻译表，对每个输入字节进行翻译。由于翻译是按字节进行的，故目标总线的逻辑宽度必须规定为 8 位。

3．同步字段(第 12～11 位)表示传送的同步方式，即 DMA 的数据传送是怎样同步的。在存贮器与存贮器之间进行数据传送时，通常采用异步的传送方式，即在通道完成了当前的传送周期(传送完一个数据)之后，若它仍然占有总线控制权，它就立即开始下一个传送周期。倘若存贮器的存取速度跟不上通道的速度，就可以利用 8284 时钟发生器的 READY 信号来扩展总线周期，即在总线周期中插入等待状态。在外部设备的速度超过通道的速度时，也可以采用这种技术，即利用等待状态来实现与外部设备之间的同步传送。当 DMA 的源是 I/O 转接口而目标是存贮器时，通常选用源同步传送方式，I/O 设备可以利用通道的 DRQ(DMA 请求)线来开始下一个传送周期，在源同步方式中，通道每收到一个 DRQ 请求信号，就执行一次 DMA 传送，然后等待下一个 DRQ 信号的到来。与此类似，当 DMA 的源是存贮器而目标为 I/O 设备时，通常选择目标同步方式，即当 I/O 设备准备好接收下一个数据(字或字节)时，它就通过 DRQ 线向通道发出请求信号。

4．源字段(第 10 位)指定 GA (或 GB)为源指针，该指针指向 $\overline{\text{DMA}}$ 传送的源，而另一个指针寄存器 GB(或 GA)则为目标指针，它指向传送的目标。

5．封锁字段(第 9 位)控制 DMA 传送过程中处理器的总线封锁($\overline{\text{LOCK}}$)信号。若封锁字段为 1，则表示在传送过程中，总线封锁信号保持有效，也就是说，在源同步的传送中，从收到

15						7			0
F	TR	SYN	S	L	C	TS	TX	TBC	TMC

F 功能字段

00 端口→端口
01 存贮器→端口
10 端口→存贮器
11 存贮器→存贮器

TR 翻译字段

0 不翻译
1 翻译

SYN 同步字段

00 无同步
01 源同步
10 目标同步
11 备用

S 源字段

0 GA 指向源
1 GB 指向源

L 封锁字段

0 无封锁
1 传送期间加封锁

C 加链字段

0 不加链
1 加链：TB 的优先权提升为级别 1

TS 单次传送字段

0 无单次传送
1 单次传送

TX 外部信号结束字段

00 非外部信号结束
01 EXT 有效时结束；位移量=0
10 EXT 有效时结束；位移量=4
11 EXT 有效时结束；位移量=8

TBC 字节结束字段

00 非字节结束
01 BC=0 时结束；位移量=0
10 BC=0 时结束；位移量=4
11 BC=0 时结束；位移量=8

TMC 屏蔽比较结束字段

000 非屏蔽比较结束
001 比较条件符合时结束；位移量=0
010 比较条件符合时结束；位移量=4
011 比较条件符合时结束；位移量=8
100 无用
101 比较条件不符合时结束；位移量=0
110 比较条件不符合时结束；位移量=4
111 比较条件不符合时结束；位移量=8

图 9.30 通道控制寄存器 CC 的格式

第一个 DMA 请求信号开始，直到通道进入结束程序为止，$\overline{LOCK}$ 信号都保持有效；若为目标同步方式，则从第一次取数据（它发生在第一次 DMA 之前）开始，一直到通道进入结束程序时为止，$\overline{LOCK}$ 信号一直在起作用。

6．加链字段（第 8 位）在 DMA 传送过程中没有什么用处。只是在该位为 1 时，它可以把通道程序的优先级别提高（提高为 1 级），这在前面已经讨论过了。

7．单次传送结束字段（第 7 位）是控制 DMA 传送结束的一种手段。当单次传送结束字段置 1 时，就表示使通道在执行完一个传送周期（即传送完一个字节或字）之后，立即恢复执行通道程序，在这种情况下，其它的传送结束条件都不再起作用，单次传送适用于和速度较慢的外部设备之间的通信，也常用在传送时需要翻译或比较的情况之中。

CC 寄存器中剩下的三个字段（第 6～0 位）用于在不采用单次传送结束方式时，告诉通道何时该结束传送。这三个字段可以单独使用，也可以结合起来使用。

外部终止信号通过通道的外部结束信号线 EXT 可使传送结束，这时，外部设备是结束当前 DMA 传送的请求者；若选用字节数作为结束传送的条件，则当字节数寄存器 BC 的内容为 0 时，通道就使当前 DMA 传送结束；如果采用屏蔽比较的结果作为结束传送的条件，则将根据 CC 寄存器中屏蔽比较结束（TMC）字段的设置情况，在发现某个传送中的字节和屏蔽的结果相等（或不相等）时，通道就将停止传送，若规定要求进行翻译处理（TR 字段为 1），则上述

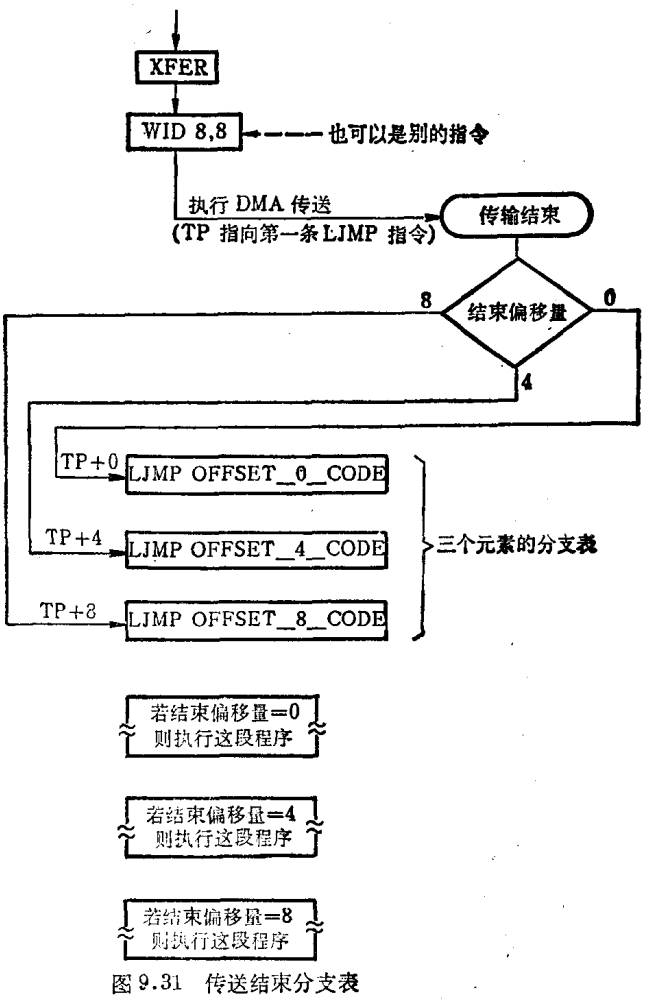


图 9.31　传送结束分支表

的比较操作就应当对翻译之后的字节数据进行。

当 DMA 传送结束时，通道将把结束偏移量加到任务块指针上去，从而形成新的任务块指针，然后通道就从程序中新指针所指的地方开始执行通道程序。所谓的结束偏移量可以取 0、4 或 8 为值，单次传送时的结束偏移量为 0。图 9.31 给出了利用加结束偏移量的方法，从分支表中不同的入口地址处开始执行通道程序的框图。

下面，我们用一个例子来说明分支表的应用。假设某次 DMA 传送的结束条件有两种，一种是传送完 80 个字节数据，另一种是在遇到回车字符的 ASCII 码 0AH 之时。也就是说，不管哪种情况先出现，都将结束当前的 DMA 传送。但是它们对应的结束偏移量是不相同的，假如先把 80 个字节传送完，通道就从分支表的 TP + 0 地址处开始执行通道程序；如果在没传送到 80 个字节时就遇到回车字符(先碰到回车符的 ASCII 码 0DH)，通道就从分支表的 TP + 4 处开始执行程序。为此，我们应当把通道控制寄存器的 TS 字段设置为 0，把 TBC 字段置为 01，而把 TMC 字段置为 010，并且要把 80 装入寄存器 BC，还要把 FF0DH 装入 MC 寄存器。在 TP + 0 和 TP + 4 中各装上一条 4 个字节长的无条件转移指令，以使通道能转去执行不同结束条件所需要的子程序。倘若传送只有一种结束条件，且其对应的结束偏移量为 0，则无需使用分支表。对于传送结束时需要无条件执行的代码，可以直接放在 XFER 指令的下面，单次传送也为这种情况。

然而应该看到，有可能出现两三个结束条件同时得到满足的情况。例如在上例中，如果第 80 个字符就是第一个回车符。通道就确认其结束条件为对应的结束偏移量最大的那个，因此，在这种情况下，通道程序应从 TP + 4 处恢复执行。

§9.4.3 DMA 传送

8089 的传送指令 XFER 使通道在执行完 XFER 后面的一条指令之后，进入 DMA 传送工作方式。对于某些外部设备控制器来讲，它们是在收到最后一个参数或命令之后就立即开始传送的，比如说 8271 软盘控制器就是这样，在这种情况下，为了给通道一些准备时间，应当在 XFER 指令的后面送出最后一个参数；如果采用的不是上述的这种控制器，则应在 XFER 指令的后面向外设控制器发出一个“启动”命令。如果需要进行的是存贮器与存贮器之间的传送，则在指令 XFER 的后面，除了改变 GA、GB 或 GC 寄存器的内容的指令之外，其它指令均可使用。但在指令 XFER 的后面，不能使用 HLT 指令，因为倘若使用 HLT 指令，则会出现 DMA 传送正在进行而通道的 BUSY 标志被清除的不合理现象。

图 9.32 给出了一个 DMA 传送周期的执行情况，一次完整的 DMA 传送就是不断地重复这种周期一直到遇到结束条件为止。该图是为了说明 DMA 传送的一般过程而故意加以简化的，特别是在修改源和目标指针(GA 和 GB)时，实际操作要比图中表示的复杂得多。值得注意的是，如果所设定的结束条件不可能出现，或者忘掉了在 CC 中设置结束条件，就会使传送开始后不会结束，因此，保证传送最后会结束是程序员的责任。

如果传送采用的是源同步方式，那么通道就将等待同步设备发出 DRQ 信号，在某个通道处于这种等待状态时，另一个通道可以自由地运行。通道取字节或字这一操作，取决于 GA 或 GB 中的源地址以及总线的逻辑宽度，表 9.11 表明了对于地址和总线宽度的各种组合，通道进行存取操作的情况。假如目标是在 16 位的逻辑总线上，而源是在 8 位的逻辑总线上，并且是

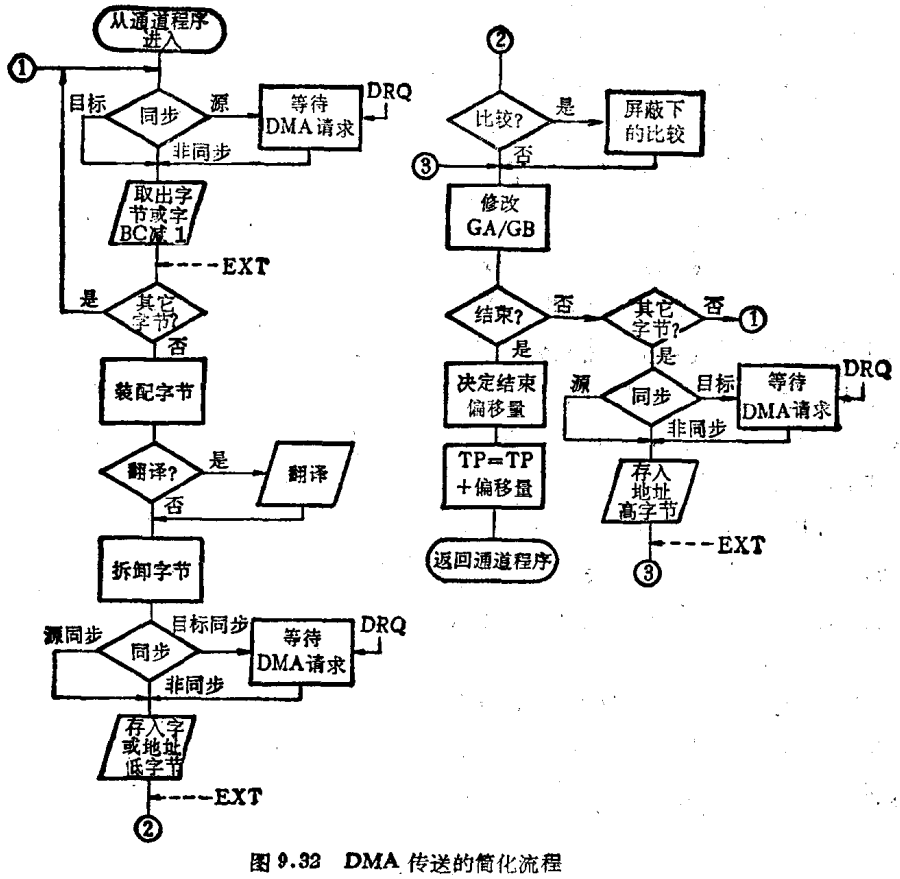


图 9.32 DMA 传送的简化流程

向偶数地址传送，那末通道就应当连续取两个字节，并在通道中把它们装配成一个字，最后送给目标。每取一次数据，通道就根据所取得的是字节还是字来修改 BC 的数值(字节减 1，字减 2)。通道在每个传送周期之后都要查看 EXT 的状态，如果预定执行两个取周期，但在头一个取周期完成后收到了 EXT 信号，这样也就不再执行第二个取周期。假如在 CC 寄存器当中规定通道对数据进行翻译的话，则应当在取完一个数据之后对之进行翻译处理。倘若已经装配好一个字或从源取来一个字，但因为目标总线是 8 位的或需传送到奇数地址中去，那么就必须把一个字拆成两个字节，分两次把该字存入目标。如果传送采用目标同步方法，则通道在执行存数周期之前必须等待 DRQ 信号。表 9.11 说明了逻辑总线宽度和奇偶地址的各种组合，也隐含地给出了为了传送一个字各自所需要执行的总线周期数目。当总线的逻辑宽度为 16 位时，

表 9.11 DMA 传送的装配与拆卸

地址 (源→目标)	逻辑总线宽度 (源→目标)			
	8→8	8→16	16→8	16→16
偶→偶	B→B	B/B→W	W→B/B	W→W
偶→奇	B→B	B→B	W→B/B	W→B/B
奇→偶	B→B	B/B→W	B→B	B/B→W
奇→奇	B→B	B→B	B→B	B→B

注：B=每一总线周期中存或取一字节
W=每一总线周期中存或取一个字
B/B=2个总线周期中存或取2个字节

除了第一个周期和最后一个周期有可能存/取一个字节数据之外，通道每次都存/取一个字。

通道在执行完第一个存贮周期之后，再次对 EXT 采样，倘若 EXT 为有效的话，通道就不再执行第二个存贮周期；如果在寄存器 CC 中把屏蔽/比较的结果设置为传送的结束条件，则低字节就必须与 MC 寄存器的屏蔽数值进行比较，当比较的结果符合设定的结束条件时，通道也将拒绝继续执行两个连续周期中的第二个周期。在上述这两种情况中，都多取出了一个数据字节，这一点将在寄存器 BC 的数值中得到反映。接下去就是对寄存器 GA 和 GB 进行修改，实际上，只有当它们指向存贮器单元时，其指针才加 1，若指向 I/O 转接口，则其内容在整个传送过程中保持不变。

如果在当前传送周期中出现了结束传送的条件，通道就将停止传送，然后，通道根据结束条件，从 CC 寄存器中得到相应的结束偏移量，并将其与 TP 相加形成通道程序恢复执行的入口地址。假如没有出现结束条件，并且通道中还有一个数据等待送出去，那么通道就存放该数据，当然，若有必要的话，在存数之前还要等待 DRQ 信号的到来，在数据送出去之后，通道还要适当修改源指针和目标指针的内容，并且再次检测结束条件。

DMA 传送可能使寄存器 BC、GA 或 GB 的内容发生变化。这是因为，在每次 DMA 传送之后都将修改 BC 寄存器中的内容（减 1 或减 2），如果寄存器 GA 或 GB 是指向存贮器的指针，则每传送一次也将修改 GA 或 GB 的内容，以使其指向下一个存贮器单元。因此，假如这些寄存器的原始内容在传送之后还要用到，则在执行传送指令 XFER 之前，应该先把它们保存到存贮器当中。对于存贮器与 I/O 转接口之间或存贮器与存贮器之间的传送来讲，程序可以通过检查指针寄存器的内容来确定 DMA 传送的最后一个字节的地址，而所传送的字节总数则可根据下式算出：

（最后一个字节的地址）-（第一个字节的地址）+1

对于 I/O 转接口与 I/O 转接口之间的 DMA 传送而言，传送的字节总数可从寄存器 BC 的原始数值减去最终数值得出，但需要满足下面两个条件：

1．BC 的初值＞BC 的终值

2．在一个传送周期完成之前，传送过程不能由于屏蔽比较条件或外部条件而结束。

一般来说，除了上面所述的一些应用之外，程序不再使用传送之后的 GA、GB 及 BC 的内容，其主要原因就在于：这些寄存器的内容还要受到多种其它因素的影响。因此，在开始一次 DMA 传送之前，都应重新设置这些寄存器。

§9.5 8089 处理器的控制和监督

本节专门讨论 8089 与 CPU 之间的相互作用，主要讨论这样几个问题：CPU 是如何初始化 8089 的、后来又是怎样向通道发出命令的、通道又是如何向 CPU 请求中断的？内容还包括通道的 DMA 控制信号以及供外部设备监督 8089 活动用的一些状态信号。

§9.5.1 8089 的初始化

在指派 8089 的通道去执行 I/O 任务之前，必须先初始化（也可称为预置）8089。所谓初始化 8089，实际上就是把所给定的系统环境通知 8089，这里的系统环境包括物理总线宽度、请

求/批准方式以及通道控制块的地址等内容。

初始化过程是从复位/(RESET)8089 开始的，当 8089 接到复位信号时，正在进行的一切操作都将停止，但该信号不影响寄存器的内容，更具体地来讲，在 8089 加电后第一次接到复位信号时，所有寄存器的内容都是未定的；对于其它的复位信号来讲，除了通道命令寄存器 CC 中的加链位被置 0 之外，其它寄存器的内容都保持不变。8089 的初始化过程如图 9.33 所示，其中的初始化控制块是由 CPU 在系统空间中为 8089 的初始化所准备的，初始化控制块的格式如图 9.34 所示，它由三个部分组成，即由系统结构指针 SCP、通道控制块 CB 和系统结构块 SCB 所组成。通道控制块 CB 通常是在 RAM 当中，而 SCP 和 SCB 可以位于 RAM 当中，也可以在 ROM 中。

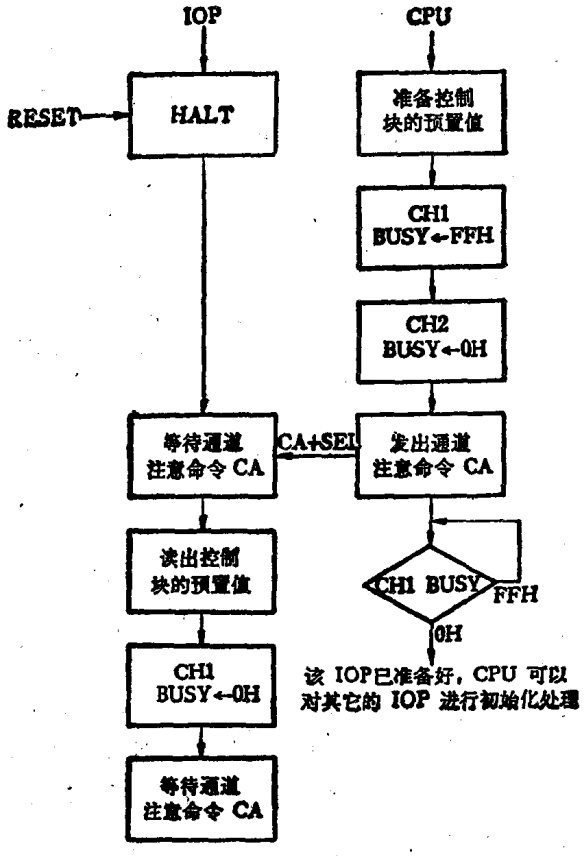


图 9.33 8089 的初始化流程

CPU 通过给通道 1(SEL 为低电平)或通道 2(SEL 为高电平)发出通道注意信号而开始初始化过程。CPU 通常是把 8089 的两个通道当作存贮器空间或 I/O 空间的两个相邻单元来访问的，若通道按 I/O 进行编址，就用输出指令 OUT 发出通道注意命令；若是按存贮器进行编址的，则用访问存贮器的指令(如 MOV 等)来发出通道注意命令。

如果欲使 8089 成为一个主设备，就应当把 SEL 置为低电平，否则，即当 SEL 为高电平时，则表示 8089 是一个从设备。在初始化过程当中，8089 对于其它的通道注意命令既不加以锁存，也不对之进行任何处理。主设备与从设备之间的主要差别就在于它们获得总线控制权的途径不同，假如 8089 是主设备，它就能够立即获得总线的控制权；倘使它是一个从设备，那么它就应当通过 $\overline{RQ}/\overline{GT}$ 向 CPU (本地方式)或另外的 8089 (远程结构方式)请求使用总线。一旦 8089 获得总线，它就在假定系统总线宽度为 8 位的前提下，到系统存贮器单元 FFFF6H 中

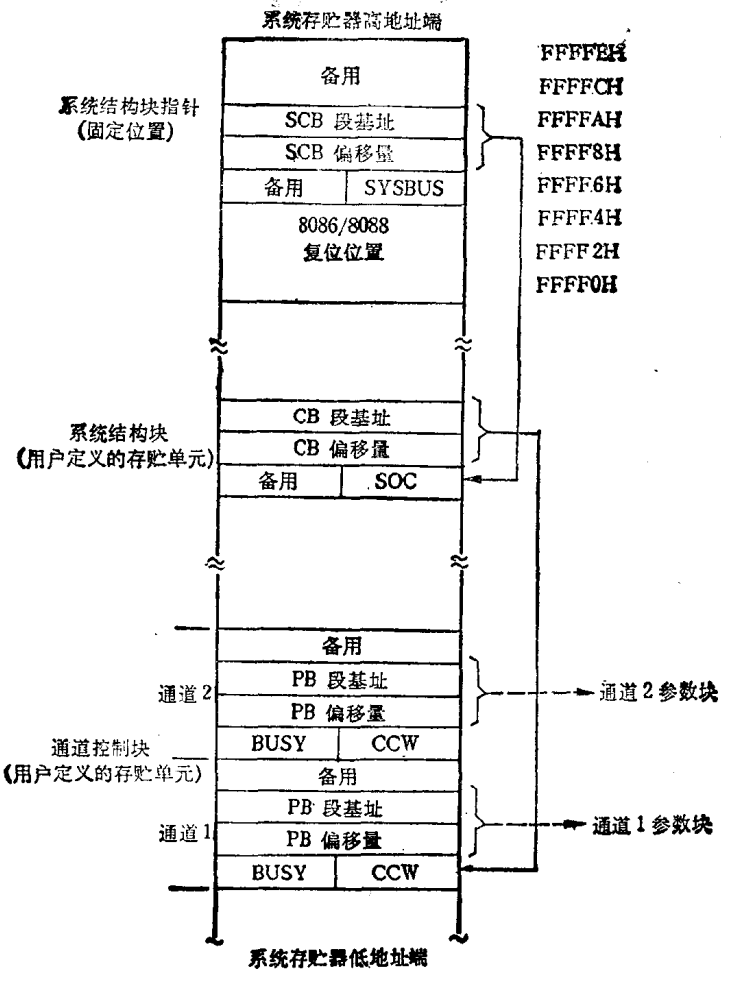
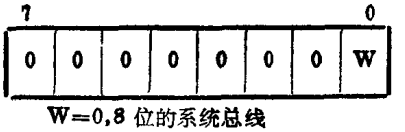


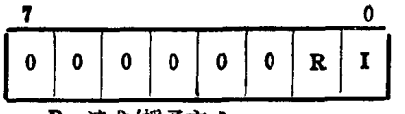
图 9.34 初始化控制块

读出 SYSBUS 字节，SYSBUS 的格式如图 9.35 所示，它把系统总线的实际宽度告诉 8089，即当第 0 位等于 0 时，表示系统总线的实际宽度为 8 位；而当第 0 位等于 1 时，则其实际宽度为 16 位。接着，8089 就从单元 FFFF8H 中读出系统结构块 SCB 的地址，它是由两个字组成的标准指示器，8089 把段基址左移 4 位再与偏移量相加就得到了 SCB 的实际地址。从这个地址当中，8089 就可以读出系统操作命令 SOC，SOC 这个字节告诉 8089 I/O 总线的宽度和请求/批准的方式(见图 9.36)。



W=0,8 位的系统总线
W=1,16 位的系统总线

图 9.35 字节 SYSBUS 的格式



R=请求/授予方式
I=0,8 位 I/O 总线
I=1,16 位 I/O 总线

图 9.36 字节 SOC 的格式

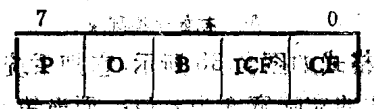
然后，8089 读出指向通道控制块的双字指针，并把它变换成 20 位的实际地址且存入内部寄存器当中，该内部寄存器只能在初始化过程中装入地址，而不能为通道程序所访问。因此，除了对 8089 重新进行初始化之外，通道控制块 CB 在执行通道程序时是不能移动的。在把 CB 的地址装入之后，8089 就把通道 1 对应的 CB 中的 BUSY 标志置 0，CB 中的其余部分在初始化过程当中不需要读出或修改，它们只有在进行通道调度时才起作用。

在初始化过程开始之后，CPU 就一直监视着 CB 中通道 1 的 BUSY 标志，以便确定初始化过程结束的时机，也就是说，一旦 BUSY 标志被置 0，就表示初始化过程已告结束，从而，CPU 也就可以给通道分配任务，当然，此时 CPU 也可以开始对另一个 8089 进行初始化处理。由于每个 8089 都有各自的 CB，所以 CPU 必须在初始化下一个 8089 之前，给出 CB 的位置，修改 SCB 中的指针。当然也可以利用多个 SCB，让每个 SCB 分别指向各自的 CB，在这种情况下，就要求 CPU 在初始化下一个 8089 之前，修改 SCP 中的指针。由此可见，在具有多个 8089 的系统中，最好把 SCB 或 SCP 放在 RAM 当中，也可以把两者都放在 RAM 当中，以提高灵活性。当所有 8089 的初始化全告完成时，CPU 就可以把 SCB 所占空间用作别的目的。

§9.5.2 通道命令

在初始化完成之后，任何通道注意信号都被解释成是给通道 1 或通道 2（根据 SEL 确定是给谁的）的命令。在 §9.3 当中，我们已经指出过，通道注意命令能否立即得到认可与两个通道的当前活动有关，但无论如何，总应当把通道注意命令锁存起来，以便一旦其优先权得到许可，就执行通道注意程序。

当通道认可了通道注意(CA)命令时，它就把 CB 中相应的 BUSY 标志置为 FFH，但这并不能阻止 CPU 发出其它的 CA 信号，所以 BUSY 标志实际上只是反映了一种状态信息。而通



CF　命令字段
000　修改 PSW
001　开始执行 I/O 空间的通道程序
010　备用
011　开始执行系统空间的通道程序
100　备用
101　恢复暂停的通道操作
110　暂停通道操作
111　停止通道操作
ICF　中断控制字段
00　对中断无影响
01　中断已经响应过，撤消中断请求
10　允许中断
11　禁止中断
B　总线加载限制
0　没有总线加载限制
1　总线加载限制
P　优先权位

图 9.87　通道命令字格式

道对 CA 的响应也就是从系统存贮器中读取各种所需要的控制信息，可见，CPU 在发出 CA 命令之前，必须先在系统存贮器中准备好这些控制信息。

在把通道的 BUSY 标志置为 FFH 之后，通道就从 CB 中读出通道命令字 CCW，然后执行这条由 CPU 所设置的命令。通道命令字的格式如图 9.37 所示。

CCW 中的命令字段 CF(第 2 位到第 0 位)共有八种状态，所对应的通道命令如图 9.38 所示。当 CF 的值是 010 或 100 时，通道的响应是无法预测的，即它们不代表任何特定的通道命令，是两个保留值；如果 CF 的值是 000，则表示这是一条修改 PSW 的命令，它可以修改 PSW 的总线加载限制位的优先权位，此外它还可以让 CPU 来控制通道内所出现的中断，这条命令对通道没有其它影响；CF 等于 001 和 011 是两条启动程序命令，两者之间的差别仅在于它们对 TP 标志位的影响不同，若 CF = 001，则通道就把该标志位置成 1，表示通道程序驻留在 I/O 空间，而当 CF = 011 时，标志位就置 0，表示通道程序存放在系统空间当中。通道把双字参数块指针变成 20 位的实际地址，并将其装入 PP 寄存器中，再把双字任务块指针装入 TP 寄存器，并检 CCW 中 ICF、B 和 P 字段的要求修改 PSW，然后从 TP 所指向的指令开始执行通道程序。

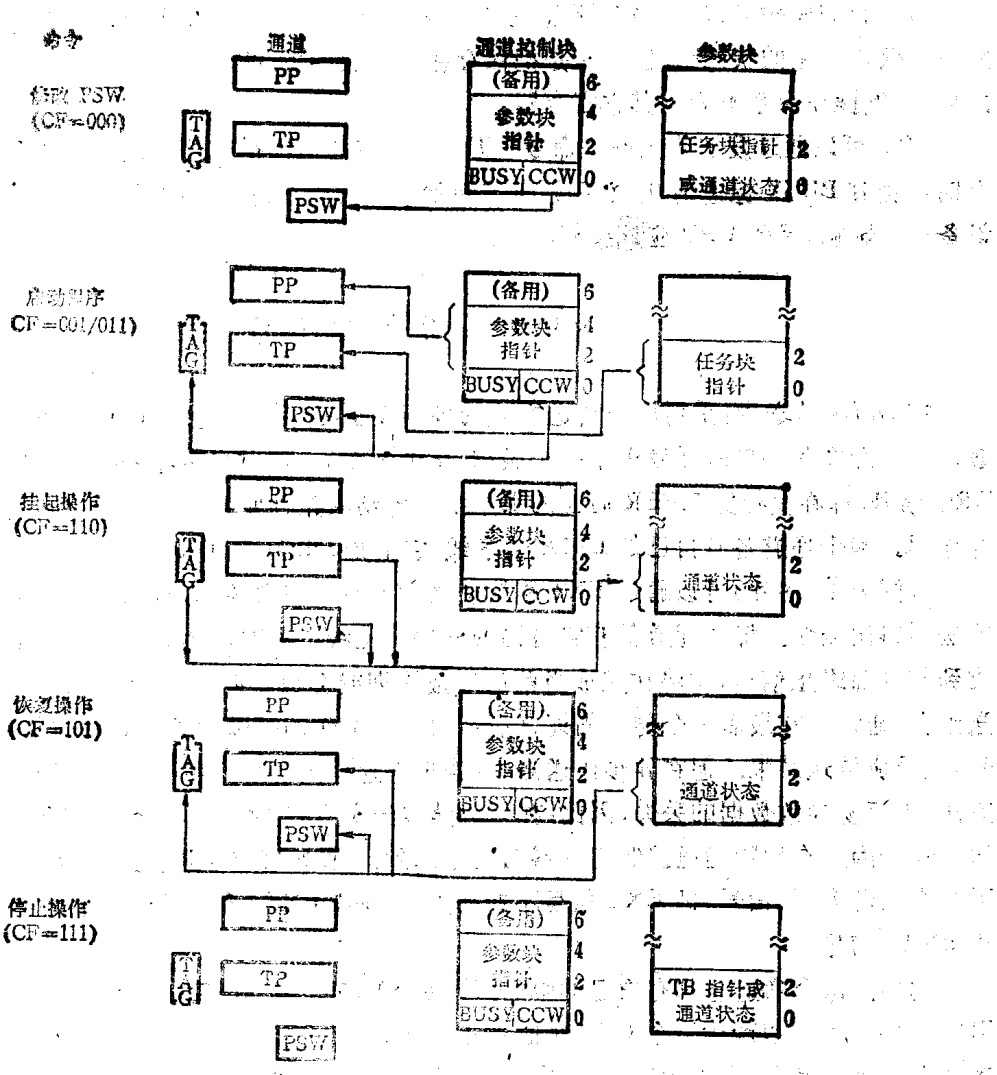


图 9.38 各种通道命令

不论通道是在执行通道程序还是在进行 DMA 传送，CPU 总可以通过把 CF 置成 110，而使得当前的通道操作暂停(或挂起)．通道在接到挂起命令之后，就把它的当前状态(TP、标志位和 PSW)保存到参数块的开头两个字中(见图 9.23)，并清除它的 BUSY 标志。执行挂起操作应注意这样几个问题：

1．由于是把通道暂停时的状态保存在原来存放通道程序首地址的双字指针单元当中，所以，倘若不重新建立该双字指针，在恢复执行时，就不可能从通道程序的起点开始执行，而只能从被挂起处(断点)开始执行。

2．TP 是执行挂起操作所必须保存的唯一寄存器，但是，假如通道启动另一个通道程序，则其它寄存器(包括 PP)都有可能被重写。幸好，在一般的情况下，挂起操作只是用来暂停通道，而不等于用另一个程序来"中断"它，即不会去执行别的程序。

3．DMA 传送的挂起不会影响 I/O 设备，也就是说，I/O 设备并不知道 DMA 传送已被暂停，它仍然象没暂停以前那样工作。比如说，它可以向通道发出 DMA 请求，只是通道此时不能响应，因为通道正处于暂停状态。因此，当操作恢复执行时，I/O 设备的状态也许与暂停时的状态不一样。

被挂起的操作可以通过把 CF 置为 101 来恢复执行，该恢复操作命令的主要功能就是恢复前面保护起来的状态信息，也就是把保存在参数块中的 TP、标志位和 PSW 送回。除了 TP 之外，恢复操作命令不影响其它寄存器。

CPU 可以通过发出停止操作命令(CF＝111)使通道的操作停止。通道在收到停止操作命令时，把其 BUSY 标志清 0，然后处于空闲状态。但需注意，DMA 传送的停止操作会对 I/O 设备产生影响，因此 CPU 应该作好准备。

§9.5.3 几个控制信号

在 DMA 传送中，采用同步传送方式的外围设备(芯片)通过发出 DRQ(DMA 请求)信号，表示它已经准备好接收或发出下一个数据(字节或字)。通道只有在 DMA 传送期间才能认可 DRQ 信号，即在执行完 XFER 后面的那条指令到结束条件发生之前这段时间内，DRQ 信号才起作用。两个通道各有自己的 DMA 请求线，它们是 DRQ1 和 DRQ2.

外围设备(芯片)可以通过 EXT 引脚使当前 DMA 传送停止，对同步设备来讲，这种结束方法用得比较多。每个通道都有它自己的外部终止线，分别记为 EXT1 和 EXT2。通道一旦收到其外部终止信号，就将在完成当前的取数周期或存数周期时，停止 DMA 传送。在同步传送当中，通常是在最后一个传送周期之后，由同步工作的外围设备(芯片)向通道发出 EXT 信号，从而使传送结束。但在异步传送当中，若采用这种由外部产生的终止信号来使传送结束的方法，则容易导致数据的丢失，因此，在多数异步传送方式当中，往往不用 EXT 信号来结束传送。这一因素在设置通道控制寄存器时要考虑到。假如通道不是在进行 DMA 传送，那它就不会去认可 EXT 信号，即 EXT 信号不起作用。倘若 EXT1 和 EXT2 同时发生，那么就先认可 EXT1 信号.

此外，每个通道也都有自己的中断线，即 SINTR1 和 SINTR2。通道程序通过执行一条 SINTR 指令就可以产生一个 CPU 中断请求，但是，执行 SINTR 指令能否使 SINTR 线变为有效还要取决于 PSW 中的中断控制位。具体讲就是：若中断控制位为 1，就允许通道发出中断

请求，从而，执行一条 SINTR 指令也就可以使相应的中断请求线成为有效；但如果中断控制位为 0，那么执行 SINTR 就不能产生相应的中断请求信号，即通道的中断请求是被禁止的。由此可见，PSW 中的中断控制位可起到屏蔽中断的作用。CPU 还可以通过向通道发出通道命令，把 CCW 中的 ICF 字段置为 10（允许中断）或 11（禁止中断），以此对 PSW 中的中断控制位进行修改，从而达到对通道的中断请求实施控制之目的。

SINTR 信号一经发出就将保持有效，直至 CPU 用通道命令把 ICF 字段置成 01 为止，CCW 中的 ICF 字段若为 01，就表示中断已经得到响应，当通道收到该命令后，它就把 PSW 中的中断服务位清 0，并且撤消中断请求。与此类似，禁止中断命令也能清除中断服务位，且把 SINTR 线降为低电平。

8089 通过三根状态线 $\overline{S0}$～$\overline{S2}$ 上的信号，向外部设备表明处理器所开始执行的总线周期类型，表 9.12 表示各种周期类型所对应的状态信号。通常总是把这些状态线和 8288 总线控制器相连接，由总线控制器对这些信号进行译码，并产生输出信号去控制挂在总线上的部件。在远程方式结构中，总线裁决器 8289 监视着状态线 $\overline{S0}$～$\overline{S2}$，以确定何时需要访问系统总线。

表 9.12 状态信号 S0～S2

$\overline{S2}$	$\overline{S1}$	$\overline{S0}$	总线周期的类型
0	0	0	从 I/O 空间取指令
0	0	1	从 I/O 空间取数据
0	1	0	把数据存入 I/O 空间
0	1	1	无用
1	0	0	从系统空间取指令
1	0	1	从系统空间取数据
1	1	0	把数据存入系统空间
1	1	1	没有执行总线周期

状态线 $\overline{S3}$～$\overline{S6}$ 用于指示当时的总线周期是否为 DMA 周期，以及该总线周期是由哪个通道所执行的，如表 9.13 所示。但需注意，当 8089 不是运行于总线周期时，这四根状态线所反映的是前已执行的最后一个总线周期的类型，例如在 8089 处于空闲状态，或在执行内部周期而不需使用总线时，情况就是如此。

表 9.13 状态信号 $\overline{S3}$～$\overline{S6}$

S6	S5	S4	S3	总线周期
1	1	0	0	通道 1 的 DMA 周期
1	1	0	1	通道 2 的 DMA 周期
1	1	1	0	通道 1 的非 DMA 周期
1	1	1	1	通道 2 的非 DMA 周期

§ 9.6 8089 对多处理器的支持

近年来，随着 LSI 和微机技术的迅猛发展，系统的性能价格比大大提高，使得应用多个处理器来满足系统实时性能要求的系统愈来愈普遍，许多系统设计工程师都认识到采用多处理技术对于改进系统的实时响应能力、提高系统的可靠性及模块化程度具有相当大的潜力。

多重处理的主要思想就是将待处理的问题划分成许多较小的任务块，再把这些任务块分给不同的处理器，由这些处理器相互协作、共同完成整个任务，也就是化整为零、各个击破思想在计算机中的应用。但是，设计者怎样才能获得这种多处理器系统呢？是否需要单独地设计各个处理器呢？即使有了多个处理器，又如何使得各处理器之间同步工作，并避免或解决资源竞争的问题呢？

通过前面的介绍，我们已经接触到了 8086、8088 CPU，对 8089 I/O 处理器也有了一个基本的了解，所以，现在是讨论把它们连成多处理器系统的时候了。

8089 作为 8086 系列中的一个专用 I/O 处理器，它能够共享 CPU 的资源，为了与 CPU 或另一个 8089 共享局部总线，在其内部具有使用局部总线的片内仲裁逻辑；8289 总线裁决器负责系统总线的仲裁；8089 的 TSL（封锁后检测并置位）指令使 8089 能借助于信号灯，与其它处理器一起共享某个资源，例如共享缓冲存贮器；在 DMA 传送期间，8089 还可以封锁系统总线，以保证传送不受其它处理器争用总线的干扰。在远程方式结构当中，8089 在电性能上是与 Intel 的多总线结构兼容的，这就意味着可以把 8089 的高速 I/O 能力用于由 8080/8085 所组成的系统当中。

我们知道，8089 与 CPU 共享系统总线，还可以与另一个 8089 及 CPU 共享其 I/O 总线，但对于总线来讲，它每次只能由一个处理器驱动，所以，当两个或两个以上的处理器公用总线时，系统就必须为之提供仲裁机构，以确定该由哪个处理器掌握总线控制权。下面就来介绍一下 8089 可用的总线裁决机构，以及它们的适用范围。

把 8089 与 8086、8088 或另一个 8089 直接相连时，就用各处理器内的 $\overline{RQ}/\overline{GT}$（请求/批准）引脚信号来决定对局部总线的使用。在本地结构当中，可以用 $\overline{RQ}/\overline{GT}$ 来控制对系统总线和 I/O 总线的访问，8089 使用总线需经下面三步：

1．8089 向 CPU 发一个信号（$\overline{RQ}/\overline{GT}$ 上的一个脉冲），请求使用总线；

2．CPU 在当前总线周期结束时，给 8089 发一个 $\overline{RQ}/\overline{GT}$ 信号，表示允许 8089 使用总线；

3．8089 在用完总线时，再向 CPU 发一个信号（还是通过 $\overline{RQ}/\overline{GT}$），表示 8089 释放总线的控制权。

前面曾经提到，在对 8089 进行初始化处理时，要对请求/批准方式加以选择，即从 8089 的两种请求/批准方式中选出一种来。当 8089 的 $\overline{RQ}/\overline{GT}$ 引脚与某个 CPU 的 $\overline{RQ}/\overline{GT0}$ 或 $\overline{RQ}/\overline{GT1}$ 相连时，就必须规定为方式 0。可见，上面给出的三步也是方式 0 的使用总线过程。当然，在一个 8089 的 $\overline{RQ}/\overline{GT}$ 引脚与另一个 8089 的 $\overline{RQ}/\overline{GT}$ 引脚相连时，也可以规定为方式 0。在方式 0 中，当 8089 与 CPU 一起使用时，CPU 为主设备，8089 为从设备；当两个 8089 一起使用时，其中的一个 8089 为主设备，另一个为从设备，主/从关系是在初始化过程中形成的。开始是主设备占有总线，并且一直保持到从设备请求使用总线时为止，从设备发出请求之后，一旦主设备"空闲"，主设备就把总线控制权让给从设备，但需注意，这里的所谓空闲对 CPU 来讲是指当前总线周期的结束，而对主设备为 8089 的情况来说，空闲是指它的两个通道都不在执行程序或在等待 DMA 请求信号。在从设备的总线请求获准之后，它就开始使用总线，直到它的两个通道都为空闲时才释放总线，把总线控制权归还给主设备。在方式 0 当中，主设备不能请求从设备归还总线。

请求/批准方式 1 只可用来对使用一条专用 I/O 总线的两个 8089 进行仲裁（图 9.8）。在方式 1 中，总是一个 8089 为主设备，另一个为从设备，这里的主设备/从设备仅仅表示开始时

总线的占有情况，一旦初始化之后，各 8089 就将根据总线的需要和占用情况，决定是该请求还是应响应。两个处理器随时都可以互相请求使用总线，一旦占有总线的处理器满足下面的两个条件，它就将把总线控制权让给发出请求的那个处理器：

1．完成一条非加链的通道程序指令；

2．由于程序暂停或完成了一个同步的传送周期(通道在等待 DMA 请求)，而使通道进入空闲状态。

但是，当通道正在执行加链的通道程序、DMA 结束程序、通道注意程序或 DMA 传送等高优先级操作时，就将阻止占用总线的 8089 把总线让给发出请求的 8089。方式 1 中的总线交换顺序为：

1．请求使用总线的处理器发出一个 $\overline{RQ}/\overline{GT}$ 脉冲，表示请求使用总线；

2．当时占有总线控制权的处理器通过 $\overline{RQ}/\overline{GT}$ 引脚发一信号作为回答信号，表示请求得到批准；

3．倘若让出总线的处理器要求总线立即归还，则应在批准应答脉冲发出后的两个时钟之后，再给 $\overline{RQ}/\overline{GT}$ 一个脉冲。

两种 $\overline{RQ}/\overline{GT}$ 工作模式之间的主要区别为：当两个处理器都在动作时，总线在两个处理器之间的切换频率不同。从一般的意义上讲，本地方式结构中，8089 $\overline{RQ}/\overline{GT}$ 操作方式应为模式 0；远程方式中的两个 8089 之间以采用 $\overline{RQ}/\overline{GT}$ 模式 1 较为适宜。模式 1 中总线的切换过程可表示为：……8089 甲请求——8089 乙响应——8089 乙请求——8089 甲响应……，由此可见，在模式 1 中，每个 8089 都能获得总线控制权，并且将保持总线控制权直到另一个 8089 发出总线请求信号，显然，这就使得通讯机构相当灵活，并且提高了总线(I/O 总线)的利用率。

在 8089 的远程方式结构当中，为了控制 8089 对系统总线的访问，就要用到 8289 总线仲裁器，当然，CPU 也有自己的总线裁决器 8289，以控制 CPU 对系统总线的使用。在由两个 8089 或一个 8089 与一个 CPU 所构成的“远程簇”中，也可以用一个 8289 来控制簇中两个处理器对系统总线的访问。

控制 8089 使用系统总线的那个 8289 将不断地监视 8089 的状态信号，当状态表示 8089 需要一个系统总线周期，而 8089 当时又不占有系统总线时，8289 就发出总线请求信号，该请求信号与其余的 8289 所发出的对系统总线的请求信号一起，按规定的途径送至优先权判优电路，从中选出优先级别最高的 8289，使它在当时的总线周期结束之后，获得系统总线。对优先级的规定是多种多样的，但在多数系统中，8089 总有比 CPU 更高的总线优先权。倘若 8289 没能使它所控制的处理器获得总线，它就把总线的状态置为“没有准备好”，与之对应，该处理器的时钟发生器就在其总线周期内插入等待状态，从而把 8089 的周期扩充到获得总线时为止。

从前面的讨论中可以看出，由于 CPU 是根据 8089 的请求转让总线的，其过程可表示为：8089 请求——CPU 响应——8089 完成，最后，CPU 重新获得总线。所以，8089 实际上比 CPU 具有更高的优先权，因而一个或两个本地的 8089 就有可能垄断总线，而排斥掉了“主”设备 CPU。当然，倘若 8089 的操作具有严格的时间要求，这种垄断也许是完全合理的；但是，也应该看到，这种垄断现象也有可能是由性能要求不高的、优先级别较低的通道程序所引起的。为了解决这一问题，给 8089 提供了总线加载限制机构，即在执行一般的通道程序(不加链的)过程中，CPU 可以通过在 CCW 中设置总线加载限制位，从而达到限制通道对总线的使用之目

的。在总线加载限制位为1时，通道每128个时钟周期就只能执行一条指令，这是通过一个7位计数器的递减实现的。采用了这种限制手段之后，就使得8089对总线的使用减少到可用总线周期的3%～25%，从而给CPU留出了很多空闲的总线周期。这种限制手段常用于本地结构当中，但也能有效地把它用于远程结构当中，特别是在执行位于系统存贮器中的通道程序时，使用这一手段就更为有效，但总线加载限制对加链的通道程序、DMA传送、DMA结束程序和通道注意程序没有任何影响。

与8086/8088一样，8089也有一个可由软件设置的总线封锁信号引脚$\overline{\text{LOCK}}$，8089的$\overline{\text{LOCK}}$输出端通常是接到总线裁决器的$\overline{\text{LOCK}}$输入端。当$\overline{\text{LOCK}}$信号有效时，总线裁决器就不会把总线释放给任何别的处理器，而不管这些处理器的优先权是多高。在执行指令TSL(封锁时检测并置位)期间，通道将自动地封锁总线，即置总线封锁信号$\overline{\text{LOCK}}$为有效；如果通道控制寄存器CC的第9位(传送时封锁总线字段)为1，则8089在其通道执行DMA传送期间，就将发出$\overline{\text{LOCK}}$信号，即若为同步传送，则从认可第一个DMA请求信号DRQ时起，使$\overline{\text{LOCK}}$信号成为有效，若传送为异步的，$\overline{\text{LOCK}}$信号就将在整个传送过程中保持有效（因为在异步传送过程中没有空闲周期），只有在通道开始执行DMA结束程序时，才撤消$\overline{\text{LOCK}}$信号。

总线封锁使传送能在尽可能短的时间内完成，并为通道排它地使用总线提供了可能。一旦某个通道获得总线控制权，并在封锁了总线的情况下开始传送，那么该通道实际上也就相当于成为总线上优先级别最高的处理器。

§9.7 8089的指令系统与寻址方式

§9.7.1 8089的指令系统

8089的指令系统共包括53条指令，按其功能，我们可以把这53条指令分为五类，即数据传送指令、算术运算指令、逻辑和位处理指令、程序转移指令、处理器控制指令。下面我们将分别对这些指令进行讨论，简单介绍一下各条指令的功能及其在通道程序中的应用。但要记住，该指令系统对存贮器单元的地址与I/O设备地址是不加区别的，也就是说，凡是可以访问存贮器中的字节或字操作数的指令，就也一定能够读/写I/O设备。

一、数据传送指令

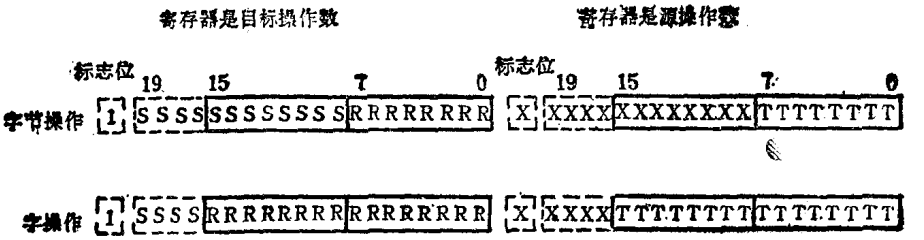
数据传送指令用于在存贮器与通道寄存器之间、存贮器与存贮器之间进行数据传送，其中包括传统的字节和字传送指令，也包括把地址装入指针寄存器的专用指令。

1．MOV(dest,src)

MOV指令把字节或字从源操作数送到目标操作数，即执行动作DEST←(SRC)。它又有四种形式：

MOV	传送字变量
MOVB	传送字节变量
MOVI	传送字立即数
MOVBI	传送字节立即数

图9.39说明了这些指令对寄存器操作数的影响，请注意，当把指针寄存器用作目标操作数时，



注：T：传送到目标操作数的位
R：本位将被源操作数的相应位所取代
S：所传送数值最高位的符号扩展位
X：无效位，它可取任意值
1：无条件置 1

图 9.39 MOV 指令中的寄存器操作数

指令就将其标志位无条件地置 1，所以，MOV 指令可用来把 I/O 空间的地址装入指针寄存器。

2．MOVP(dest,src)

指示器传送指令 MOVP 在指针寄存器和存贮器之间传送一个物理地址。若源操作数为指针寄存器，则将其内容与其标志位结合起来变换成一个物理地址指示器(见图 9.27)；若源为存贮器单元，则将三个字节变换成一个 20 位的物理地址和一个标志位，并分别把它们装入指针寄存器(由目标操作数给出)及其相应的标志位当中。通常把 MOVP 指令用来保存和恢复各指针寄存器的内容。

3．LPD(dest,src)

LPD 是一条把双字变量装入指针寄存器的指令，它把双字指针变换成一个 20 位的物理地址，并将结果装入目标指针寄存器。执行该指令后，指针寄存器的标志位肯定为 0，表示该指针指向系统空间。这种类型的指令共有两条：

LPD　　把双字变量装入指针寄存器

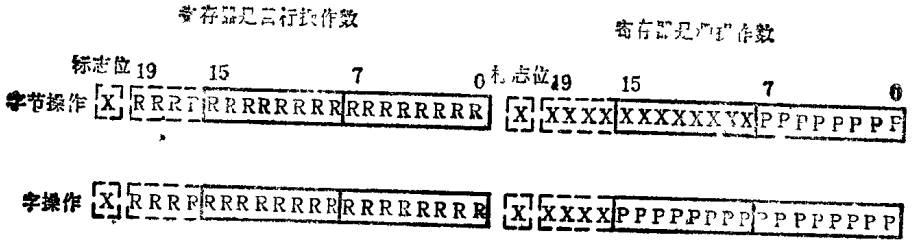
LPDI　　把双字立即数装入指针寄存器

8086/8088 可以把它的 1 兆字节存贮空间中任一单元的地址以双字指针的形式送至通道程序，通道程序通过 LPD 指令把该单元的地址装入指针寄存器，就可以访问这个单元了。

二、算术运算指令

算术指令把各种操作数解释为不带符号的 8 位、16 位或 20 位 2 进制数，带符号的数值是以标准补码的形式来表示的，其最高位是符号位("0"为正，"1"为负)。然而，值得注意的是处理机无法发现进入符号位的溢出位，这一问题必须由用户软件负责解决。

8089 进行 20 位算术运算采用的是这样一种方法：首先符号扩展字节或字操作数到 20 位，即字节操作数的第 7 位送到内部寄存器的第 8～19 位，字操作数的第 15 位送到内部寄存器的第 16～19 位。符号位的扩展不会影响操作数的值。在完成算术操作后，得到一个 20 位的结果，然后将结果送到目标操作数的所在单元中，高位部分被舍去，使之正好与目标的有效空间相符合，但这样也就检测不出在结果的高位部分所反映的进位或借位。当然，如果目标寄存器的长度比源操作数的位数大，则进位信息将在寄存器的高位部分得到体现。



图中 X：无效位，它在操作中可取任意值
R：本位由操作结果所取代
P：本位不与操作

图 9.40 算术运算指令中的寄存器操作数

图 9.40 说明当寄存器被规定为源或目标操作数时，算术运算指令是怎样对待它们的：

1. ADD(dest,src)

源操作数与目标操作数相加，结果送至目标操作数。共有四种这样的加法指令：

ADD　　字变量相加

ADDB　　字节变量相加

ADDI　　字立即数相加

ADDBI　　字节立即数相加

2. INC(dest)

目标操作数加"1"，有两条指令：

INC　　字加"1"

INCB　　字节加"1"

3. DEC(dest)

目标操作数减"1"，也有两条这样的指令：

DEC　　字减"1"

DECB　　字节减"1"

三、逻辑和位操作指令

逻辑指令包括一般的布尔操作"与"(AND)、"或"(OR)、"非"(NOT)，它们都是按位运算的。与此类似，为了置位或清除存贮器单元或 I/O 设备寄存器中的某一位，8 80 还提供了四条位操作指令。对于逻辑操作来讲，20 位目标寄存器的高 4 位是没有定义的，但应注意，当某个寄存器被指定为字节逻辑操作的目标寄存器时，它的第 8～15 位就将用字节操作结果的第 7 位进行重写(如图 9.41 所示)。

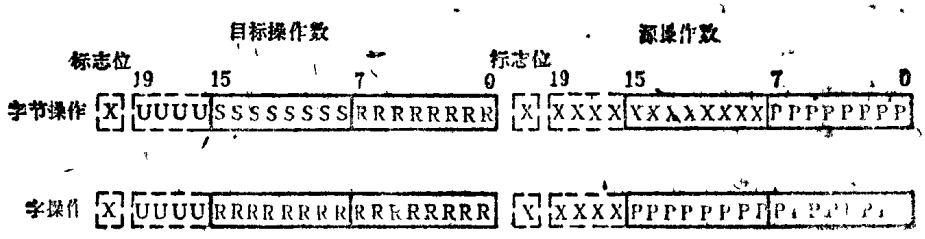


图 9.41 逻辑指令中的寄存器操作数

图中的X表示运算中没用到的位，U为操作后结果不确定的位，R表示参与操作并由操作结果所取代的位，S为结果的符号扩展位，P代表参与操作但不发生改变的2进位。

1．AND(dest,src)

AND指令把两个操作数逻辑"与"起来，并将结果送到目标操作数。逻辑"与"操作定义为：若两个操作数的对应位都为"1"，则得到的结果位也为"1"，否则，只要其中有一位不是"1"，那么所得到的结果位就为"0"。AND指令有下面几种形式：

AND　　字变量的逻辑"与"

ANDB　　字节变量的逻辑"与"

ANDI　　字立即数的逻辑"与"

ANDBI　　字节立即数的逻辑"与"

当需要清除设备寄存器中的某几位而保留其余位不变时，就可以用AND指令。例如，把8位寄存器和EEH进行逻辑"与"操作，就清除且仅清除掉了寄存器的第0位和第4位，其它各位仍然保留原来的数值。

2．OR(dest,src)

OR指令对两个操作数进行逻辑"或"运算，得到的结果送到目标操作数当中。对于逻辑"或"运算来讲，若两个操作数所对应的两个位中至少有一个为"1"，则结果的相应位就为"1"，而只有在参加运算的两位皆为"0"时，结果的对应位才取"0"值。OR指令也有四种形式：

OR　　字变量逻辑"或"

ORB　　字节变量逻辑"或"

ORI　　字立即数逻辑"或"

ORBI　　字节立即数逻辑"或"

OR指令可用来有选择地把设备寄存器中的多位置"1"。例如，8位寄存器与30H进行逻辑"或"操作，则结果的第4位和第5位均被置为"1"，其它各位保持不变。

3．NOT(dest/dest,src)

NOT指令完成操作数的取反操作。操作数字段中的"/"代表"或者"，即说明NOT指令有两种类型的操作数，如果带的是一个单操作数，就用取反所得到的结果取代原值；若是两个操作数，就对源操作数的各位求反，结果送至目标操作数(必须为寄存器)，而源操作数保持原值不变。NOT指令除了具有两种操作数形式之外，还为字和字节取反操作分别提供了指令助记符：

NOT　　字逻辑"非"

NOTB　　字节逻辑"非"

若操作数为正数，则在NOT指令后面加上一条INC指令，就将得到操作数的补码。

4．SETB(目标,位选)

指令SETB把目标操作数中的某一位无条件地置为"1"，其中的目标操作数必须是一个存贮器字节。若位选值为0，则把目标操作数的最低位置"1"；当位选值取7时，则被置位的将是目标操作数的最高位。借助于SETB指令，可以很方便地置位8位设备寄存器的某一位。

5．CLR(目标,位选)

这也是一条位操作指令，它与SETB的操作可说是完全相同，只是它将所选择的位无条件地清"0"，而不是象SETB那样置"1"。

四、程序转移指令

通道程序的执行顺序是由寄存器TP所控制的。每执行一条指令，都将把指令的长度加到TP当中，使之指向下一条指令。但对于程序转移指令来讲，情况就不同了，它是把一个带符号的偏移量与TP相加，从而改变程序的执行顺序。其中的偏移量包含在程序转移指令当中，它可以是8位的，也可为16位，偏移量是用2的补码表示的，最高位为符号位，若符号位为0，则偏移量为正，故为往前跳指令；若符号位等于1，则表示偏移量是负值，即指令为往回跳指令。这里的偏移量是相对于跳转指令的结束点而言的，若为8位偏移量，则跳转范围是－128～＋127个字节，若偏移量是16位的，则跳转范围就扩大到－32768～＋32767个字节中的任一单元。通常我们也把偏移量为8位的转移指令称为短跳转指令，而把偏移量为16位的转移指令叫做长跳转指令。

每种程序转移指令都有两个指令助记符，一个以字母"L"开头，另一个不以"L"开头。如果助记符以字母"L"开头，则说明它是一个长跳转指令，故不管目标的远近，都用一个16位的偏移量来表示为了到达目标所需跳过的距离(字节数目)；若指令助记符不以字母"L"开头，则由ASM-89汇编语言根据一定的规则，建立8位或16位的偏移量。

程序转移指令所采用的相对寻址技术导致了下面两个重要结论：第一，它有力地支持了编码的位置独立性，即其代码可在存贮器中移动且仍然能够正确执行，这里只有一个限制，即这种移动必须把整个程序当作一个单位来进行，其理由也很简单，因为只有这样，才能使转移指令和转移的目标之间的距离保持不变；第二，在设计大型的通道程序时，必须记住寻址范围具有一定的局限性。

1．CALL/LCALL(TPsave,target)

CALL指令是一个调用子程序的指令。为了使子程序能够返回到CALL下面的那条指令，有必要把执行CALL指令时TP的值保存起来。CALL指令把TP及其标志位保存在操作数TPsave当中，这个操作数必须是一个实际地址变量。然后就转移到目标地址处继续运行，目标地址是由目标操作数的偏移量加上TP形成的。利用一条MOVP指令把TPsave的内容装入TP，就可以使子程序返回到CALL下面的那条指令。

应该注意的是8089用于调用子程序或过程的机构没有8086/8088的相应部分那样完善。最根本的差异在于8089没有机内硬件堆栈，而是靠用程序建立堆栈机构，用基址寄存器作为堆栈指针。当然也应该看到，既然通道程序不受中断支配，所以对于大多数通道程序来说也就不需要使用堆栈。

2．JMP/LJMP(target)

JMP指令是一条无条件转移指令，它使控制转移到目标地址。由于执行该指令时，没有把任务指针TP保存起来，所以程序不能返回到JMP后面的那条指令。

3．JZ/LJZ(src,target)

在源操作数为0的情况下，执行JZ指令将使程序转移到目标程序单元；否则程序将继续顺序执行。对应于源操作数为字或字节这两种情况，"为0则转"指令有下面两种形式：

JZ/LJZ：如果字操作数为0，则跳转/长跳转。

JZB/LJZB：如果字节为0，则跳转/长跳转。

假如源操作数是一个寄存器，则仅测试寄存器的低16位是否为0，可以不考虑寄存器的

其余高位部分；若要测试寄存器的低 8 位，则先要清除掉第 8～15 位，然后再用操作数为字的 JZ/LJZ 指令。

4．JNZ/LJNZ(src,target)

JNZ 指令与 JZ 指令类似，所不同的只是指令 JNZ 是在源操作数不等于 0 时实现转移。它也有两种形式：

JNZ/LJNZ：如果字操作数不为 0 则跳转/长跳转。

JNZB/LJNZB：若字节操作数不为 0 则跳转/长跳转。

5．JMCE/LJMCE(src,target)

如果源操作数（存贮器中的一个字节）等于屏蔽过的 MC 寄存器中的低位字节，则指令 JMCE/LJMCE 就将使现行程序转移到目标单元。本指令可用来对 8 位设备寄存器中的若干位进行检测。

6．JMCNE/LJMCNZ(src,target)

若源操作数与 MC 寄存器的值进行比较，其结果为不相等的话，则转移到目标单元。其它操作与 JMCE 指令相同。

7．JBT/LJBT(src,bit-select,target)

JBT 指令对源操作数(由 src 指出)中的某一位(由 bit-select 指出)进行测试，若被测试位的值为 1，则程序转移到目标单元。其中，源操作数必须是存贮器或 I/O 设备寄存器中的一个字节，位选值的取值范围是 0～7，第 0 位指最低位。这条指令可以用于测试 8 位设备寄存器中的某一位，如果转移的目标就是 JBT 指令本身，则该指令所完成的操作实际上也就成为“等待到被选择位为 0”，即只有在被选位等于 0 时，程序才能够继续执行下去。

8．JNBT/LJNBT(src,bit-select,target)

该指令所执行的操作与 JBT 类似，不同点只是它是在被选择位等于 0（而不是 1）的条件下发生转移。

五、处理器控制指令

通道程序通过使用这些处理器控制指令，就能够控制 8089 I/O 处理器的一些硬设备，例如，对 $\overline{\text{LOCK}}$ 和 SINTR1—2 引脚的输出信号实施控制，选择逻辑总线宽度以及启动 DMA 传送等。

1．TSL(dest,set-value,target)

TSL 指令的操作流程如图 9.42 所示，利用 TSL 指令，可以为多处理器系统中共享资源的分配提供控制手段，这里实际上是用了一个信号灯。从图 9.42 中也可以看出，若目标(target)单元为 TSL 指令本身，就将重复执行这条指令，直到信号灯(dest)呈 0 状态时为止。由此可见，当通道需要使用某个共享资源时，在通道程序中安排这样一条 TSL 指令，就会使它直到所需资源被释放时才能继续执行下去。

2．WID(src-width,dest-width)

WID 指令为 DMA 传送设定逻辑总线宽度，它能改变 PSW 中的第 0 位和第 1 位的内容。指令中的操作数可定为 8 位或 16 位，但不能超过对应总线的实际宽度。处理机刚刚复位时，总线的逻辑宽度未经定义，即逻辑宽度是不确定的，因此，在执行 DMA 传送之前，必须先执行 WID 指令。在执行了 WID 指令之后，逻辑宽度就将保持为指令中所设定的值，直到再次执行

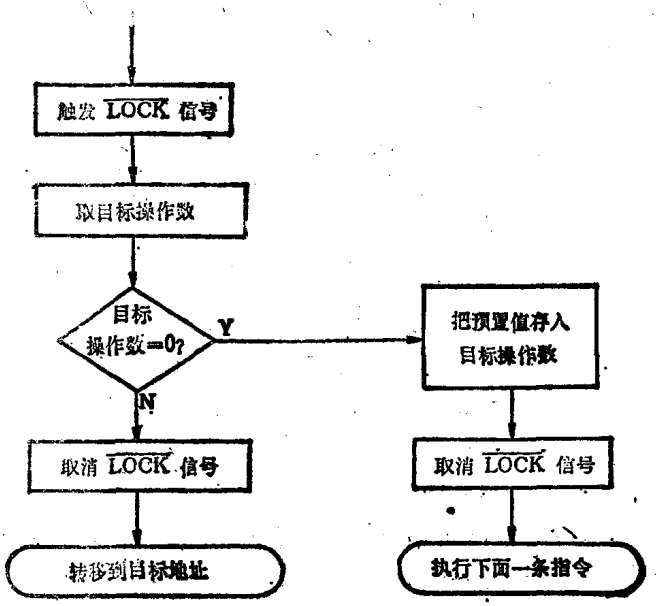


图 9.42 TSL 指令的操作流程

一个 WID 指令或处理器复位时为止。

3. XFER

XFER 指令为通道进行 DMA 传送做准备，通道在执行完该指令的下面一条指令之后就进入 DMA 传送方式。在同步传送方式中，紧跟在 XFER 后面的指令往往是命令同步设备做好准备(例如发出一个"启动"命令或该设备所需要的最后一个参数)。除了修改 GA、GB 或 GC 的指令之外，其它的任何一条指令(包括 NOP 和 WID)都可以跟在 XFER 指令的后面。

4. SINTR

SINTR 指令使 PSW 中的中断服务位置"1"。假如 PSW 中的中断控制位为 1，那么这条指令就激活相应通道上的中断请求信号 SINTR；但如果中断控制位为 0，则表示通道禁止中断，故尽管中断服务位为 1，也不能激活中断请求信号(SINTR1 或 SINTR2)。通道程序可以用这条指令去中断 CPU。

5. NOP

NOP 指令只消耗几个时钟周期，而没有其它作用，即它执行的是空操作。在定时循环当中，经常使用该指令。

6. HLT

HLT 指令用于结束通道程序，它使通道在清除了它的 BUSY 标志后进入空闲状态。

§9.7.2 8089 的寻址方式

8089 指令的操作数可以存放在寄存器中，也可以放在系统和 I/O 地址空间中，还可以由指令本身提供(立即数)。在系统和 I/O 地址空间中的操作数或在存贮器单元中，或在 I/O 设备的寄存器中，并可用四种不同的方式来对它们进行寻址。下面主要介绍通道是如何处理各种操作数以及怎样利用寻址方式来计算地址的。

在许多指令当中，寄存器都可以用作源或目标操作数，从一般的意义上讲，对寄存器进行

操作的指令要比对立即数或存贮器操作数进行操作的指令短一些，且要快一些。

立即操作数是包含是指令当中的数据，而不是在寄存器或存贮器当中。立即数的长度可为 8 位或 16 位，它只能用作源操作数，且必须是常数。

尽管通道可以直接访问寄存器和立即操作数，但是，8089 若要访问系统或 I/O 空间的操作数则必须通过总线。要做到这一点，首先就要求 8089 计算出操作数的地址，即计算出所谓的有效地址(EA)，程序员可以根据需要，使用某种寻址方式(四种寻址方式之一)来计算操作数的地址。

系统空间中的操作数具有 20 位有效地址，而 I/O 空间中操作数的有效地址为 16 位，这些地址都是不带符号的数，它们表示从地址空间的起点到操作数的低字节之间的距离。由于 8089 看不到它与 8086/8088 共享的系统空间的分段结构，所以 8089 的有效地址就等价于 8086/8088 的物理地址。

8089 的所有寻址方式都要用到某个指针寄存器的内容，因此，根据该寄存器标志位的状态，就可以确定所寻址的操作数是处于系统空间中还是处于 I/O 空间中。如果操作数位于 I/O 空间中(标志位 = 1)，则在计算有效地址时就将忽略指针寄存器的第 16～19 位。下面分别介绍 8089 的四种寻址方式。

1．基址寻址方式

在基址寻址方式中、有效地址可以直接从指针寄存器 GA、GB、GC 或 PP 中获得(图 9.43)。在使用这种寻址方式时，如果在指令执行之前修改基址寄存器的内容，则该指令就能访问不同的单元。通过前面对指令的介绍，大家应该清楚 LPD、MOV、MOVP 及算术运算指令都可以用来改变基址寄存器的内容。

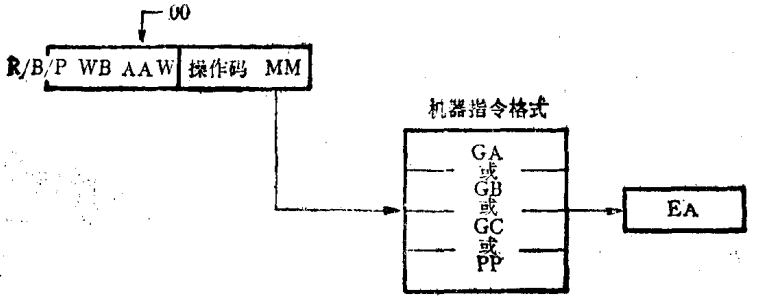


图 9.43 基址寻址方式

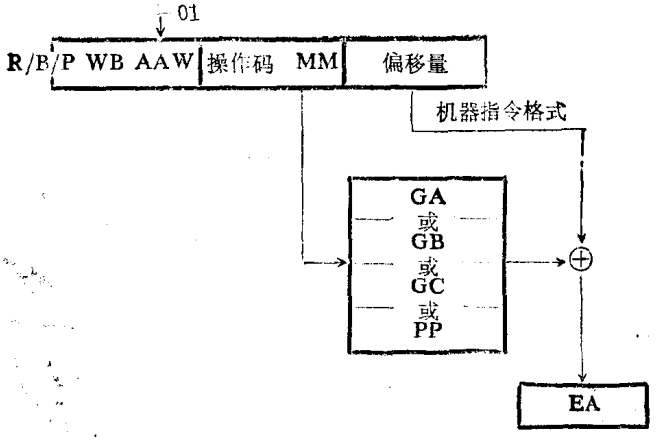


图 9.44 偏移寻址方式

2．偏移寻址方式

在偏移寻址方式中，指令含有一个 8 位的不带符号的偏移量，把该偏移量与基址寄存器的内容相加，就得到了操作数的有效地址（见图 9.44）。

采用偏移寻址方式，可以很方便地对结构中的元素进行寻址。如图 9.45 所示，让基址寄存器指向某结构块的基地址，即指向结构的第一个地址单元，然后，就可用不同的偏移值来寻址该结构内的元素。通过利用改变基地址的方法，就能使同一个结构在存贮器的其它地方浮动出现。对于参数块中单元的寻址，常采用偏移寻址方式，这里的基址寄存器为 PP。

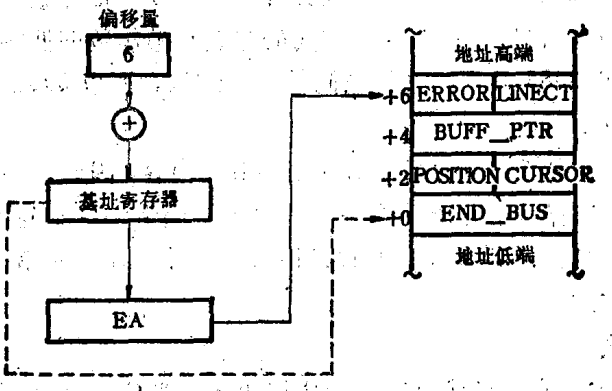


图 9.45 偏移寻址方式举例

3．变址寻址方式

变址寻址中的有效地址是由寄存器 IX 的内容（不带符号的数）加上基址寄存器的内容形成的，如图 9.46 所示。变址寻址方式常用来访问数组中的元素，就象图 9.47 所表示的那样，让基址寄存器指向数组的起始单元，然后用变址寄存器 IX 选择某个数组元素，可见 IX 起着一个数组下标的作用。当 IX 的内容为(i－1)时，表示选择字节数组的第 i 个元素；若要访问字数组的第 i 个元素，IX 的内容则应为((i－1)*2)。

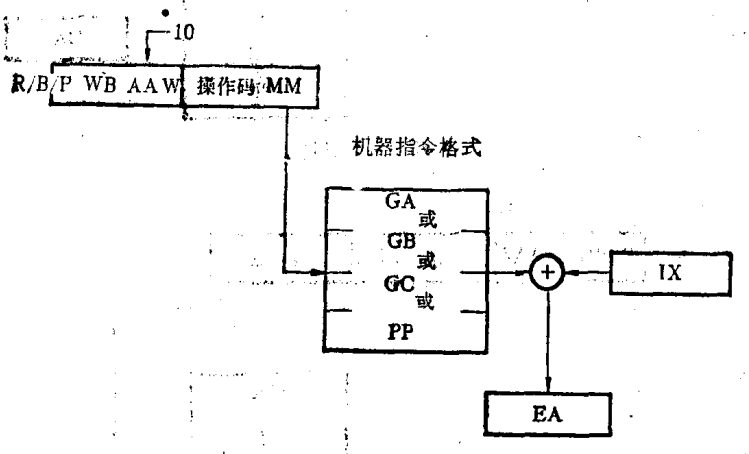


图 9.46 变址寻址方式

4．变址递增寻址方式

在这种变址寻址方式中，有效地址也是由 IX 的内容和基址寄存器的内容相加所形成的，这一点是与上面介绍的变址寻址方式一致的，但是，利用变址递增寻址方式计算出有效地址之后，变址寄存器 IX 就将递增，如图 9.48 所示。若寻址的是字节操作数，则 IX 递增 1；若为字

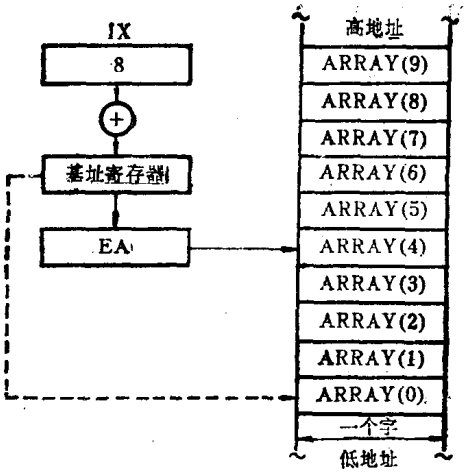


图 9.47　变址寻址方式举例——字数组寻址

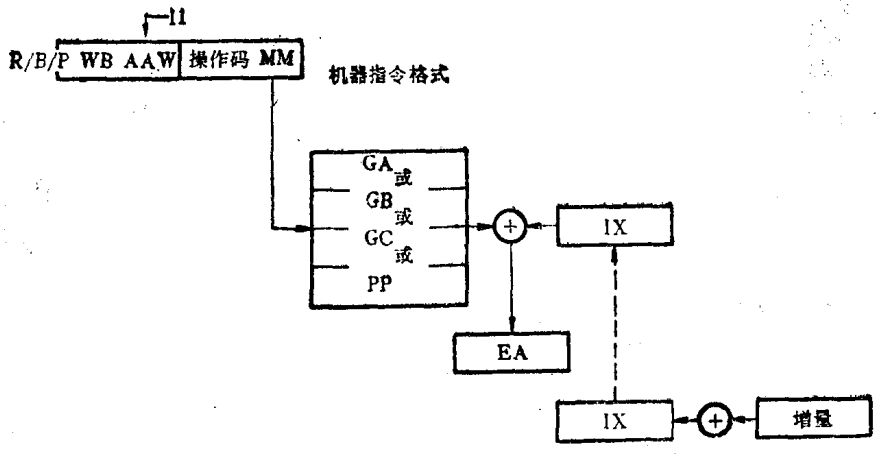


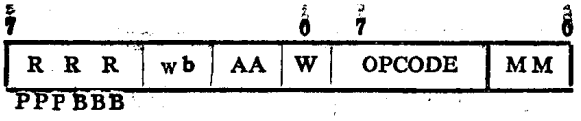
图 9.48　变址自增寻址方式

操作数，IX 就递增 2；在执行 MOVP 指令时，IX 就将递增 3。这种寻址方式对于以“步进方式”通过数组的各个元素的寻址是非常有用的，例如在求数组元素之和的循环当中，采用这种寻址方式将是甚为方便的。

§9.7.3　8089 指令系统小结

在 §9.7.1 中，我们曾经对 8089 的每条指令分别进行过讲解，简单介绍了各条指令的功能及适用场合，由于是按功能分别进行的介绍，所以显得比较繁琐，而且也没有给出这些指令的执行时间和指令长度等一些基本信息。为了解决这些问题，下面将主要以表格的形式列出指令系统的一些参考数据，并将给出各条指令的编码格式，目的就是想给读者掌握 8089 的指令系统、使用 8089 的指令提供一些参考用的工具。

8089 的任意一条指令至少要占用两个字节，指令集中相当大的一部分指令还要求使用更多的存贮器单元，以便确定操作数地址或操作数本身（立即数）。8089 的指令格式可以表示为：



下面给出指令内各字段的定义。

MM	基 址 寄 存 器
0 0	GA
0 1	GB
1 0	GC
1 1	PP

寄存器字段 RRR 在指令当中指出一个 16 位的寄存器操作数，倘若 RRR 代表的是 GA、GB、GC 或 TP，则将寄存器的最高 4 位(第 16～19 位)置成与符号位(第 15 位)相同。RRR 字段的编码是这样的：

RRR			
000	r0	GA	
001	r1	GB	
010	r2	GC	
011	r3	BC	；字节数
100	r4	TP	；任务块指针
101	r5	IX	；变址寄存器
110	r6	CC	；通道控制
111	r7	MC	；屏蔽/比较

PPP 字段用来指出一个 20 位的地址指针寄存器，其编码如下：

PPP		
0 0 0	p0	GA
0 0 1	p1	GB
0 1 0	p2	GC
1 0 0	p4	TP

在位操作指令当中，寄存器字段 RRR 将由位选择字段 BBB 所取代，由 BBB 指出指令所要处理的是哪一位。当 BBB 为 000 时，则处理第 0 位(最低位)；若 BBB 为 111，则处理第 7 位(最高位)。

Wb(字/字节)字段为 01 时，表示是一个字节立即数；若为 10，则说明有一个字立即数(两个字节)。

AA 字段指出四种操作数寻址方式，小结如下：

AA	寻 址 方 式
0 0	由 MM 字段所选择出的基址寄存器中含有操作数地址
0 1	所选择的基址寄存器的内容加上指令中给出的一个 8 位的无符号偏移量形成操作数地址．基址寄存器的内容不变
1 0	基址寄存器的内容与变址寄存器的内容相加形成操作数地址，两个寻址寄存器的内容均保持不变
1 1	寻址过程基本同上(AA=10)，只是在求得地址后，变址寄存器自动递增(8 位传送加 1，16 位传送加 2)

操作数类型字段 W(宽度)指出所选择的操作数为 1 个字节长 (W = 0)，还是两个字节长 (W = 1)。

8089 的指令除了包括上面这两个指令字节之外，还有可能需要几个附加字节。例如，寻址操作数时所需要的一个 8 位无符号偏移量(AA = 01)；转移指令中的 8 位或 16 位有符号位移量以及立即数指令中所包含的 8 位或 16 位立即数 (PILD 指令需要 32 位的立即数)。这样一些附加的字节在 8089 指令当中出现的次序为：

偏移量	立即数	转移目标的位移

表 9.14 根据指令的类型，列出了 8089 的所有指令及其相应的代码格式。

表 9.14 8089 指令及其代码表

(一) 数据传送指令(DATA TRANSFER INSTRUCTIONS)

(1) MOV：传送字变量

	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
·存贮器送到寄存器	RRR0 0AA1	1 0 0 C 0 0MM	偏移(AA=01)	
·寄存器送到存贮器	RRR0 0AA1	1 0 0 0 0 1MM	偏移(AA=01)	
·存贮器送到存贮器	0 0 0 0 0AA1	1 0 0 1 0 0MM	偏移(AA=01)	0 0 0 0 0AA1
		1 1 0 0 1 1MM	偏移 (AA=01)	

(2) MOVB：传送字节变量

·存贮器送到寄存器：	RRR0 0AA0	1 0 0 0 0 0MM	偏移 (AA=01)	
·寄存器送到存贮器：	RRR0 0AA0	1 0 0 0 0 1MM	偏移 (AA=01)	
·存贮器送到存贮器：	0 0 0 0 0AA0	1 0 0 1 0 0MM	偏移 (AA=01)	0 0 0 0 0AA0
		1 1 0 0 1 1MM	偏移 (AA=01)	

(3) MOVBI：传送字节立即数

·立即数送到寄存器：	RRR0 1 0 0 0	0 0 1 1 0 0 0 0	8 位数据	
立即数送到存贮器：	0 0 0 0 1AA0	0 1 0 0 1 1MM	偏移 (AA=01)	8 位数据

(4) MOVI：传送字立即数

·立即数送到寄存器：	RRR1 0 0 0 1	0 0 1 1 0 0 0 0	低位数据	高位数据	
·立即数送到存贮器：	0 0 0 1 0AA1	0 1 0 0 1 1MM	偏移 (AA=01)	低位数据	高位数据

(5) MOVP：传送指针

·存贮器送到指针寄存器：	PPP0 0AA1	1 0 0 0 1 1MM	偏移 (AA=01)
·指针寄存器送到存贮器：	PPP0 0AA1	1 0 0 1 1 0MM	偏移 (AA=01)

(6) LPD：用可变双字装入指示器：

PPP0 0AA1	1 0 0 0 1 0MM	偏移 (AA=01)

(7) LPDI：用双字立即数装入指示器

PPP1 0 0 0 1	0 0 0 0 1 0 0 0	低位偏移	高位偏移	低位段址	高位段址

(二)算术运算指令(ARITHMETIC INSTRUCTIONS)

(1) ADD：字变量加

·存贮器与寄存器：

RRR00AA1	101000MM	偏移 (AA=01)

·寄存器与存贮器：

RRR00AA1	110100MM	偏移 (AA=01)

(2) ADDB：字节变量加

·存贮器与寄存器：

RRR00AA0	101000MM	偏移 (AA=01)

·寄存器与存贮器：

RRR00AA0	110100MM	偏移 (AA=01)

(3) ADDI：字立即数加

·立即数与寄存器：

RRR10001	00100000	低位数据	高位数据

·立即数与存贮器：

00010AA1	110000MM	偏移 (AA=01)	低位数据	高位数据

(4) ADDBI：字节立即数加

·立即数与寄存器：

RRR01000	00100000	8位数据

·立即数与存贮器：

00001AA0	110000MM	偏移 (AA=01)	8位数据

(5) INC：字递增

·寄存器递增

RRR00000	00111000

·存贮器递增

00000AA1	111010MM	偏移 (AA=01)

(6) INCB：字节递增

00000AA0	111010MM	偏移 (AA=01)

(7) DEC：字递减

·寄存器递减：

RRR00000	00111100

·存贮器递减：

00000AA1	111011MM	偏移 (AA=01)

(8) DECB：字节递减

00000AA0	111011MM	偏移 (AA=01)

(三)逻辑和位处理指令(LOGICAL AND BIT MANIPULATION)

(1) AND：字变量"与"

·存贮器和寄存器"与"：

RRR00AA1	101010MM	偏移 (AA=01)

·寄存器和存贮器"与"：

RRR00AA1	110110MM	偏移 (AA=01)

(2) ANDB：字节变量"与"

·存贮器和寄存器"与"：

RRR00AA0	101010MM	偏移 (AA=01)

·寄存器和存贮器"与"：

RRR00AA0	110110MM	偏移 (AA=01)

(3) ANDI：字立即数"与"

立即数和寄存器"与"：

RRR10001	00101000	低位数据	高位数据

·立即数和存贮器"与"：

00010AA1	110010MM	偏移 (AA=01)	低位数据	高位数据

(4) ANDBI：字节立即数"与"

立即数和寄存器"与"：

RRR01000	00101000	8位数据

·立即数和存贮器"与"：

00001AA0	110010MM	偏移 (AA=01)	8位数据

(5) OR：字变量"或"

·存贮器和寄存器"或"：

RRR00AA1	101001MM	偏移 (AA=01)

·寄存器和存贮器“或”：

RRR00AA1	110101MM	偏移 (AA=01)

(6) ORB：字节变量“或”

·存贮器和寄存器“或”：

RRR00AA0	101001MM	偏移 (AA=01)

·寄存器和存贮器“或”：

RRR00AA0	110101MM	偏移 (AA=01)

(7) ORI：字立即数“或”：

·立即数和寄存器“或”：

RRR10001	00100100	低位数据	高位数据

·立即数和存贮器“或”：

00010AA1	110001MM	偏移 (AA=01)	低位数据	高位数据

(8) ORBI：字节立即数“或”

·立即数和寄存器“或”：

RRR01000	00100100	8 位数据

·立即数和存贮器“或”：

00001AA0	110001MM	偏移 (AA=01)	8 位数据

(9) NOT：字变量“非”

·寄存器

RRR00000	00101100

·存贮器

00000AA1	110111MM	偏移 (AA=01)

·存贮器和寄存器“非”

RRR00AA1	101011MM	偏移 (AA=01)

(10) NOTB：字节变量“非”

·存贮器

00000AA0	110111MM	偏移 (AA=01)

·存贮器和寄存器“非”

RRR00AA0	101011MM	偏移 (AA=01)

(11) SETB：置位

BBB00AA0	111101MM	偏移 (AA=01)

(12) CLR：清除

BBB00AA0	111110MM	偏移 (AA=01)

(四)程序转移指令(PROGRAM TRANSFER INSTRUCTIONS)

(1) CALL：调用

10001AA1	100111MM	偏移 (AA=01)	8 位位移量

(2) LCALL：长调用

10010AA1	100111MM	偏移 (AA=01)	低位位移量	高位位移量

(3) JMP：无条件转移

10001000	00100000	8 位位移量

(4) LJMP：无条件长转移

10010001	00100000	低位位移量	高位位移量

(5) JZ：字为零时转移

·寄存器内容为 0 时转移

RRR01000	01000100	8 位位移量

·存贮器内容为 0 时转移

00001AA1	111001MM	偏移 (AA=01)	8 位位移量

(6) LJZ：字为零时长转移

寄存器内容为零时转移

RRR10000	01000100	低位位移量	高位位移量

存贮器内容为零时转移

00010AA1	111001MM	偏移 (AA=01)	低位位移量	高位位移量

(7) JZB：字节为零时转移

00001AA0	111001MM	偏移 (AA=01)	8 位位移量

(8) LJZB：字节为零时长转移

00010AA0	111001MM	偏移 (AA=01)	低位位移量	高位位移量

(9) JNZ：字不为零时转移

寄存器内容不为零时跳转

RRR 01000	01000000	8 位位移量

存贮器内容不为零时跳转

00001AA1	111000MM	偏移 (AA=01)	8 位位移量

(10) LJNZ：字不为零时长转移

寄存器内容不为零时转移

RRR 10000	01000000	低位位移量	高位位移量

存贮器内容不为零时转移

00010AA1	111000MM	偏移 (AA=01)	低位位移量	高位位移量

(11) JNZB：字节不为零时转移

00001AA0	111000MM	偏移 (AA=01)	8 位位移量

(12) LJNZB：字节不为零时长转移

00010AA0	111000MM	偏移 (AA=01)	低位位移量	高位位移量

(13) JMCE：带屏蔽比较相等时转移

00001AA0	101100MM	偏移 (AA=01)	8 位位移量

(14) LJMCE：带屏蔽比较相等时长转移

00010AA0	101100MM	偏移 (AA=01)	低位位移量	高位位移量

(15) JMCNE：带屏蔽比较不等时转移

00001AA0	101101MM	偏移 (AA=01)	8 位位移量

(16) LJMCNE：带屏蔽比较不等时长转移

00010AA0	101101MM	偏移 (AA=01)	低位位移量	高位位移量

(17) JBT：选择位为"1"时转移

BBB01AA0	101111MM	偏移 (AA=01)	8 位位移量

(18) LJBT：选择位为"1"时长转移

BBB10AA0	101111MM	偏移 (AA=01)	低位位移量	高位位移量

(19) JNBT：选择位不为"1"时转移

BBB01AA0	101110MM	偏移 (AA=01)	8 位位移量

(20) LJNBT：选择位不为"1"时长转移

BBB10AA0	101110MM	偏移 (AA=01)	低位位移量	高位位移量

(六)处理机控制指令

(1) TSL：测试

00011AA0	100101MM	偏移 (AA=01)	8 位数据	8 位位移量

(2) WID：设置逻辑总线宽度

1SD00000	00000000

注：S=源宽度，D=目标宽度；"0"为 8 位，"1"为 16 位

(3) XFER：进入 DMA 方式

01100000	00000000

(4) SINTR：置中断工作位

01000000	00000000

(5) HLT：停止通道程序

00100000	01001000

(6) NOP：空操作

00000000	00000000

在§9.7.1对指令的介绍当中，我们在指令的操作数部分使用了一些标识符，例如JZ/LJZ(src,target)中圆括号内的src和target，尽管大多数读者都能领会由它们所代表的意思，但为了便于大家更好地理解，也为了便于后面的介绍，特对所用到的操作数标识符作如下说明(见表9.15)。

表9.15 操作数标识符的说明

标识符	用途	说明
destination/dest	数据传送、算术运算、位操作中的目标操作数	它是寄存器或存贮器单元，其中可以含有待处理的数据，也可以接收运算结果
source/src	数据传送、算术运算、位操作中的源操作数	它是指令执行中所用到的一个寄存器、存贮器单元或立即数，指令不能修改它
target	程序转移的目标	它是一个地址单元，转移成功时控制将转移到这一目标
TPsave	程序转移(调用子程序)	24位存贮器单元，用来保存下一条指令的地址
bit-select	位操作指令	确定操作数字节中待处理的位号. 0代表最右位，7表示最高有效位(最左位)
set-value	TSL	当目标操作数为0时，把它赋给目标操作数
source-width(src-width)	WID	源总线的逻辑宽度
dest-width	WID	目的总线的逻辑宽度

所有操作数都具有一定的类型，多数指令对其操作数都有一定的类型限制，类型不同，指令的动作也不一样。表9.16给出了8089指令中各种操作数类型的说明。

表9.16 操作数类型说明

标识	说明
(no operands)	无操作数
register	任一通用寄存器
ptr-reg	指针寄存器
immed 8	立即数字节(0～FFH的常数)
immed 16	立即数字(0～FFFFH中的常数)
mem 8	8位存贮器单元(字节)
mem 16	16位存贮器单元(字)
mem 24	24位存贮器单元(物理地址指示器)
mem 32	32位存贮器单元(双字指针)
label	标号：在距指令－32768～＋32767个字节的范围内
short-label	标号：在距指令－128～＋127个字节的范围内
0—7	0～7之间的常数
8/16	常数8或16

表 9.17 按照指令助记符的字母顺序列出了所有 8089 指令，从该表中可以了解到 ASM-89 的指令格式及其名称，表中还为各种操作数类型的组合，提供了相应的指令执行时间(clocks)、指令长度(所占字节数 Bytes)以及应用示例(Coding Example)。对于操作数为存贮器字的指令来讲，由于该操作数位于偶地址与位于奇地址所花费的访问时间不一样，故指令的执行时间也就不同，所以我们在这类指令的执行时间(clocks)一栏中给出了两个数值，它们分别为操作数位于偶地址/奇地址时，指令的执行时间。

表 9.17 指令系统参考数据表

ADD destination, source		Add Word Variable	
Operands	Clocks	Bytes	Coding Example
register, mem 16	11/15	2-3	ADD BC, [GA], LENGTH
mem 16, register	16/26	2-3	ADD [GB], GC

ADDB destination, source		Add Byte Variable	
Operands	Clocks	Bytes	Coding Example
register, mem 8	11	2-3	ADDB GC, [GA].N—CHARS
mem 8, register	16	2-3	ADDB [PP].ERRORS, MC

ADDBI destination, source		Add Byte Immediate	
Operands	Clocks	Bytes	Coding Example
register, immed 8	3	3	ADDBI MC, 10
mem 8, immed 8	16	3-4	ADDBI [PP+IX+].RECORDS, 2CH

ADDI destination, source		Add Word Immediate	
Operands	Clocks	Bytes	Coding Example
register, immed 16	3	4	ADDI GB, 0C25BH
mem 16, immed 16	16/26	4-5	ADDI [GB], POINTER, 5899

AND destination, source		Logical AND Word Variable	
Operands	Clocks	Bytes	Coding Example
register, mem 16	11/15	2-3	AND MC, [GA].FLAG—WORD
mem 16, register	16/26	2-3	AND [GC], STATUS, BC

ANDB destination, source		Logical AND Byte Variable	
Operands	Clocks	Bytes	Coding Example
register, mem 8	11	2-3	AND BC, [GC]
mem 8, register	16	2-3	AND [GA+IX], RESULT, GA

ANDBI destination, source		Logical AND Byte Immediate	
Operands	Clocks	Bytes	Coding Example
register, immed 8	3	3	ANDBI GA, 01100000B
mem 8, immed 8	16	3-4	ANDBI [GC+IX], 2CH

ANDI destination, source		Logical AND Word Immediate	
Operands	Clocks	Bytes	Coding Example
register, immed16	3	4	ANDI IX,0H
mem 16, immed 16	16/26	4-5	ANDI [GB+IX].TAB, 40H

CALL TPsave,target		Call	
Operands	Clocks	Bytes	Coding Example
mem24, label	17/23	3-5	CALL [GC+IX].SAVE, GET—NEXT

CLR destination, bit select		Clear Bit To Zero	
Operands	Clocks	Bytes	Coding Example
mem 8, 0-7	16	2-3	CLR [GA],3

DEC destination		Decrement Word By1	
Operands	Clocks	Bytes	Coding Example
register	3	2	
mem 16	16/26	2-3	DEC [PP].RETRY

DECB destination		Decrement Byte By1	
Operands	Clocks	Bytes	Coding Example
mem 8	16	2-3	DECB [GA+IX+].TAB

HLT (no operands)		Halt Channel Program	
Operands	Clocks	Bytes	Coding Example
(no operands)	11	2	HLT

INC destination		Increment Word by1	
Operands	Clocks	Bytes	Coding Example
register	3	2	INC GA
mem 16	16/26	2-3	INC [GA].COUNT

INCB destination		Increment Byte by1	
Operands	Clocks	Bytes	Coding Example
mem 8	16	2-3	INCB [GB].POINTER

JBT source, bit-select, target		Jump if Bit True(1)	
Operands	Clocks	Bytes	Coding Example
mem 8, 0-7, label	14	3-5	JBT [GA]. RESULT—REG, 3, DATA—VALID

JMCE source, target		Jump if Masked Compare Equal	
Operands	Clocks	Bytes	Coding Example
mem 8, label	14	3-5	JMCE [GB].FLAG, STOP—SEARCH

JMCNE source, target		Jump if Masked Compare Not Equal	
Operands	Clocks	Bytes	Coding Example
mem 8, label	14	3-5	JMCNE [GB+IX], NEXT—ITEM

JMP target		Jump Unconditionally	
Operands	Clocks	Bytes	Coding Example
label	3	3-4	JMP READ—SECTOR

JNBT source, bit-select, target		Jump if Bit Not True(0)	
Operands	Clocks	Bytes	Coding Example
mem 8, 0-7, label	14	3-5	JNBT [GC], 3, RE—READ

JNZ source, target		Jump if Word Not Zero	
Operands	Clocks	Bytes	Coding Example
register, label	5	3-4	JNZ BC, WRITE—LINE
mem 16, label	12/16	3-5	JNZ [PP].NUM—CHARS, PUT—BYTE

JNZB source, target		Jump if Byte Not Zero	
Operands	Clocks	Bytes	Coding Example
mem 8, label	12	3-5	JNZB [GA], MORE—DATA

JZ source, target		Jump if Word is Zero	
Operands	Clocks	Bytes	Coding Example
register, label	5	3-4	JZ BC, NEXT—LINE
mem 16, label	12/16	3-5	JZ [GC+IX]. INDEX, BUF—EMPTY

JZB source, target		Jump if Byte Zero	
Operands	Clocks	Bytes	Coding Example
mem 8, label	12	3-5	JZB [PP]. LINES—LEFT, RETURN

LCALL TPsave, target		Long Call	
Operands	Clocks	Bytes	Coding Example
mem 24, label	17/23	4-5	LCALL [GC].RETURN—SAV,E INIT—8279

LJBT source, bit-select, target		Long Jump if Bit True(1)	
Operands	Clocks	Bytes	Coding Example
mem 8, 0-7, label	14	4-5	LJBT [GA].RESULT, 1, DATA—OK

LJMCE source, target		Long jump if Masked Compare Equal	
Operands	Clocks	Bytes	Coding Example
mem 8, label	14	4-5	LJMCE [GB],BYTE—FOUND

LJMCNE source, target		Long jump if Masked Compare Not Equal	
Operands	Clocks	Bytes	Coding Example
mem 8, label	14	4-5	LJMCNE [GC+IX+], SCAN—NEXT

LJMP target		Long Jump Unconditional	
Operands	Clocks	Bytes	Coding Example
label	3	4	LJMP GET—CURSOR

LJNBT source, bit-select, target		Long Jump if Bit Not True(0)	
Operands	Clocks	Bytes	Coding Example
mem 8, 0-7, label	14	4-5	LJNBT [GC], 6, CR CC—ERROR

LJNZ source, target		Long Jump if Word Not Zero	
Operands	Clocks	Bytes	Coding Example
register, label	5	4	LJNZ BC, PARTIAL—XMIT
mem 16, label	12/16	4-5	LJNZ [GA+IX]. N—LEFT, PUT—DATA

LJNZB source, target		Long Jump if Byte Not Zero	
Operands	Clocks	Bytes	Coding Example
mem 8, label	12	4-5	LJNZB [GB+IX+]. ITEM, BUMP—COUNT

LJZ source, target		Long Jump if Word Zero	
Operands	Clocks	Bytes	Coding Example
register, label	5	4	LJZ IX, FIRST—ELEMENT
mem 16, label	12/16	4-5	LJZ [GB], XMIT—COUNT, NO—DATA

LJZB source, target		Long Jump if Byte Zero	
Operands	Clocks	Bytes	Coding Example
mem 8, label	12	4-5	LJZB [GA], RETURN—LINE

LPD destination, source		Load Pointer With Doubleword Variable	
Operands	Clocks	Bytes	Coding Example
ptr-reg, mem 32	20/28*	2-3	LPD GA, [PP]. BUF—START

* 若操作数的地址为偶数，则需要 20 个时钟周期，否则，需要 28 个时钟周期

LPDI destination, source		Load Pointer With Doubleword Immediate	
Operands	Clocks	Bytes	Coding Example
ptr-reg, immed 32	12/16*	6	LPDI GB, DISK—ADDRESS

* 若指令位于偶地址，则需要 12 个时钟的执行时间．否则，需要 16 个时钟的执行时间

MOV destination, source		Move Word	
Operands	Clocks	Bytes	Coding Example
register, mem 16	8/12	2-3	MOV IX, [GC]
mem 16, register	10/16	2-3	MOV [GA]. GOUNT, BC
mem 16, mem 16	18/28	4-6	MOV [GA]. READING, [GB]

MOVB destination, source		Move Byte	
Operands	Clocks	Bytes	Coding Example
register, mem 8	8	2-3	MOVB BC, [PP]. TRAN—COUNT
mem 8, register	10	2-3	MOVB [PP]. RETURN—CODE, GC
mem 8, mem 8	18	4-6	MOVB[GB+IX+], [GA+IX+]

MOVBI destination, source		Move Byte Immediate	
Operands	Clocks	Bytes	Coding Example
register, immed 8	3	3	MOVBI MC, 'A'
mem 8, immed 8	12	3-4	MOVBI [PP]. RESULT, 0

MOVI destination, source		Move Word Immediate	
Operands	Clocks	Bytes	Coding Example
register, immed 16	3	4	MOVI BC, 0
mem 16, immed 16	12/18	4-5	MOVI[GB], 0FFFFH

MOVP destination, source		Move Pointer	
Operands	Clocks	Bytes	Coding Example
ptr-reg, mem 24	19/27	2-3	MOVP TP, [GC+IX]
mem 24, ptr-reg	16/22	2-3	MOVP[GB].SAVE—ADDR, GC

NOP (no operands)		No Operation	
Operands	Clocks	Bytes	Coding Example
(no operands)	4	2	NOP

NOT destination/destination, source		Logical NOT Word	
Operands	Clocks	Bytes	Coding Example
register	3	2	NOT MC
mem 16	16/26	2-3	NOT [GA].PARM
register, mem 16	11/15	2-3	NOT BC, [GA+IX]. LINES—LEFT

NOTB destination/destination, source		Logical NOT Byte	
Operands	Clocks	Bytes	Coding Example
mem 8	16	2-3	NOTB[GA].PARM—REG
register, mem 8	11	2-3	NOTB IX, [GB]. STATUS

OR destination, source		Logical OR Word	
Operands	Clocks	Bytes	Coding Example
register, mem 16	11/15	2-3	OR MC, [GC]. MASK
mem 16, register	16/26	2-3	OR [GC], BC

ORB destination, source		Logical OR Byte	
Operands	Clocks	Bytes	Coding Example
register, mem 8	11	2-3	ORB IX, [PP].POINTER
mem 8, register	16	2-3	ORB [GA+IX+], GB

ORBI destination, source		Logical OR Byte Immediate	
Operands	Clocks	Bytes	Coding Example
register, immed 8	3	3	ORBI IX, 00010001B
mem 8, immed 8	16	3-4	ORBI[GB].COMMAND, 0CH

ORI destination, source		Logical OR Word Immediate	
Operands	Clocks	Bytes	Coding Example
register, immed 16	3	4	ORI MC, 0FF0DH
mem 16, immed 16	16/26	4-5	ORI [GA], 1000H

SETB destination, bit-select		Set Bit to 1	
Operands	Clocks	Bytes	Coding Example
mem 8, 0-7	16	2-3	SETB [GA].PARM—REG, 2

SINTR (no operands)		Set Interrupt Service Bit	
Operands	Clocks	bytes	Coding Example
(no operands)	4	2	SINTR

TSL dest, set-value, target		Test and Set while Locked	
Operands	Clocks	Bytes	Coding Example
mem 8, immed 8, short-lable	14/16*	4-5	TSL [GA].FLAG, 0FFH, NOT—ready

* 若目标操作数≠0，则需要14个时钟周期；否则，即若dest=0，则需要16个时钟周期

WID source-width, dest-width		Set Logical Bus Widths	
Operands	Clocks	Bytes	Coding Example
8/16, 8/16	4	2	WID 8, 8

XFER (no operands)		Enter DMA Transfer After Next Instruction	
Operands	Clocks	Bytes	Coding Example
(no operands)	4	2	XFER

§9.8 8089的程序设计

§9.8.1 8089汇编语言 ASM-89

通过前面的介绍，人们应该有这样一个概念，即作为I/O处理器，8089除了能够完成一般的输入/输出操作之外，它自己还具有执行程序的能力。它所执行的程序(通道程序)通常是用8089的汇编语言ASM-89编写的，用汇编语言写出的通道程序常被称为源程序模块，为了使8089能够执行这些程序，还要把源程序模块送到ASM-89汇编程序，经过汇编得到一个可浮动的目标程序模块，如果汇编过程中没有发生错误，那么8089就可以执行该程序了。

一、语句

语句是ASM-89程序的基本组成部分，即程序是由一些语句所组成的。例如，下面这些例子都是ASM-89中的语句：

```
;THIS STA TEMENT CONTAINS A COMMENT FIELD ONLY(只是个注释语句)
ADDI BC,5              ;TYPICAL ASM 89 INSTRUCTION(典型的ASM89指令格式)
  ADDI   BC,     5     ;NO"COLUMN"REQUIREMENTS(没有对准的要求)
MOV  [GA].STATUS,
```

```
&          6                ;A CONTINUED STATEMENT(写到两行的一个语句)
SOURCE     EQU GA           ;A SIMPLE ASM 89 DIRECTIVE(ASM 89 的伪指令)
LINE_BUFFER_
           ADDRESS DD       ;A LONG IDENTIFIER(该语句说明可用较长的标识符)
```

从上面的例子中可以看出，语句的格式是相当灵活的，这就为程序员提供了方便。与 ASM-86 类似，ASM-89 中标识符的长度也不得超过 31 个字符，标识符是一个字母数字串，其中也可以插入下划线(_)，这就使得比较长的名字也能较容易地读懂，从而提高了程序的可读性，例如象 WAIT_UNTIL_READY 这样的标识符，不仅机器能够识别它，而且人也能从中获得一些信息. 语句是程序的结构单位，但它本身又是由一些更基本的成分(字段或标记)所组成的，这些字段无需定位在语句的某些特定的"列"上，它们之间可以用一个或多个空隔字符分开，操作数字段中的各个操作数之间也可以插入任意个空隔字符. 假如语句很长，需要继续到下一行，那么只要在下一行的开头处写上字符"&"即可，例如上例中的

```
MOV    [GA].STATUS,
&      6
```

以分号开头的语句是注释语句，注释语句丝毫不影响程序的执行，ASM-89 不为注释语句产生任何目标代码，但在程序当中适当地加上一些注释语句，就能够大大提高程序的文本化程度，为阅读程序提供方便，这是因为它能反映出程序的逻辑关系，而这往往是指令本身所不能显式表达出来的.

ASM-89 中的语句也可以分成指令性语句和指示性语句(伪指令)两个大类. 汇编程序碰到指令性语句时，必须为之产生目标代码(机器指令). 指令性语句的一般格式为：

```
标号：   指令助记符    操作数    ;注释
```

例如语句：

```
DEMO:  JNBT    [GA].STATUS,3,DEMO    ;WAIT UNTIL READY
```

其中的标号是可有可无的，若语句带标号，那么该标号标识符的值就是存放此指令的存贮器单元的地址，带标号的语句可以作为程序转移的目标，只要转移指令把该标号用作它的目标操作数. 例如，上例中的指令 JNBT 带有标号，它有条件地转移到本身重复执行. 指令性语句中的指令助记符是不能省略的，它规定了汇编程序将要产生的机器代码. 操作数字段对有些指令来讲是必需的，还有可能出现多个操作数，但对另外一些指令来讲，它又可能被禁止出现. 注释字段总是可供选择的，但需注意，它必须以分号(;)打头.

指示性语句的最大特点就是它不产生任何机器代码，它所起的作用只是为汇编程序提供一些汇编用的信息. 其格式为：

```
名字    伪指令助记符    操作数    ;注释
```

或

```
标号：  伪指令助记符    操作数    ;注释
```

例如

```
INPUT_BUFFER:  DS    80    ;TERMINAL LINE STORED HERE
```

这个语句告诉汇编程序为之保留 80 个存贮器字节，INPUT_BUFFER 指向该保留空间的第一个单元.

指示性语句中的第一个字段为名字或标号，若用作标号，则其后面也要加上冒号(:)，不同

的是伪指令中的标号不能用作程序转移的目标，标号是可供选择的；与标号不同，名字的后面不能跟上冒号，名字也不是一个可以任意选择的项，是否需要名字要根据伪指令的要求来确定，它可能是必需的、可供选择的，也可能是被禁止出现的。指示性语句的第二个字段是伪指令助记符，汇编程序利用该字段来区分指令和伪指令，ASM-89 汇编程序共可接收 14 条伪指令。接下去是伪指令所要求的操作数，多个操作数之间用逗号或空格分开。指示性语句中也可以包括注释部分，注释也以分号(;)开头。

二、常数

常数是 ASM-89 程序中出现的一些固定值。ASM-89 中的常量也分为数字常量和串常量，在 ASM-89 的指令及伪指令当中，可以出现 2 进制、8 进制、10 进制和 16 进制的数字常量。汇编程序在进行汇编的过程中，还可以对常数进行加、减运算。但数值常量(包括算术运算的结果)必须能用 16 位 2 进制数表示，即正数不能大于 65535，负数不能小于 -32768，负数在汇编程序中是以 2 的补码表示的。2 进制形式以 B 结尾，8 进制以 Q 或 O 结尾，10 进制以 D 结尾也可不加 D，16 进制则以 H 结尾。

字符串常量是用单引号(撇号)括起来的字符序列。当作为指令操作数使用时，单引号内只能包含一个或两个字符，它们分别对应于字节指令和字指令。但当把字符串用来初始化存贮器时，串中的字符数最多可达 255，即它最多可以每次给 255 个存贮器字节置初值。常数的应用例子如下：

```
MOVBI  GA,'A'          (字符串常数)
MOVBI  GA,41H          (16 进制常数)
MOVBI  GA,65           (10 进制常数)
MOVBI  GA,65D          (10 进制常数)
MOVBI  GA,101Q         (8 进制常数)
MOVBI  GA,101O         (8 进制常数)
MOVBI  GA,01000001B    (2 进制常数)
```

上面这七个语句实际上都是等价的，其作用都相同。下面的两个语句也是等价的，它们说明了负数的补码表示方法。

```
MOVBI    GA,-5
MOVBI    GA,11111011B
```

为了使程序更为清晰，我们也可以借助于伪指令 EQU 给常数取名。例如：

```
DISK_STATUS    EQU    0FF20H
```

三、数据定义

数据定义语句为变量分配存贮器，并把变量名与该存贮器地址联系起来，以便能在指令性语句或某些伪指令中用名字来引用这些存贮单元，有些数据定义语句还能给这些单元置初始值。在 ASM-89 当中，共有四条伪指令可用来为变量保留内存空间。

伪指令 DB、DW 和 DD 分别以字节、字、双字为单位给变量分配存贮器。DB 和 DW 可以用于定义单个的字节变量和字变量，也可以用来定义字节数组和字数组。例如，一个字符串常量就可以被定义为一个字节数组：

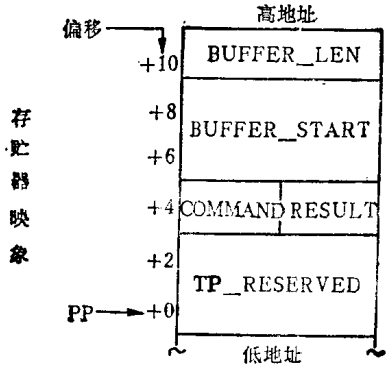
```
SIGN_ON_MSG:    DB    'PLEASE ENTER PASSWORD'
```

伪指令DD的典型应用是定义系统空间的存贮器单元地址，即定义一个双字指示器变量，利用指令LPD就可以把双字变量(地址)装入指针寄存器。伪指令DS可以为变量保留任意个字节的存贮器空间，但它不能赋给任何存贮器单元以初始值，利用该伪指令可以在RAM中开辟象缓冲区之类的保留区域。此外，还可以用它来定义物理地址指示器，而MOVP指令所用到的三字节物理地址指示器是其它三个数据定义伪指令所不能直接定义的。

举例如下：

```
ASM 89 的指示性语句                          存贮器中的内容
ALPHA:  DB   1              ;01H
        DB   -2             ;FEH(2 的补码)
        DB   'A','B'        ;4142H
BETA:   DW   1              ;0100H
        DW   -5             ;FAFFH
        DW   'AB'           ;4241H
        DW   400,500        ;2410F401H
        DW   400H,500H      ;00040005H
gamma:  DW   BETA           ;OFFSET OF BETA ABOVE,FROM BEGINNING
                            ;OF PROGRAM(BETA 的偏移量)
DELTA   DD   GAMMA          ;ADDRESS OF GAMMA
                            (GAMMA 的地址,包括段分量及偏移量)
ZETA:   DS   80             ;80 BYTES UNINITIALIZED
                            (80 个字节的内存空间,设置初值)
```

在ASM-89当中还可以对结构进行定义，所谓结构是指一些有关变量的集合，其中的变量被称为结构元素或结构元，结构定义给出了这些元素的名字和相对位置。但是定义一个结构时并不为该结构分配存贮器，这一点是与前面所介绍的几条伪指令所不同的。图9.49表示出如何用结构来定义参数块，以及通道程序中是怎样使用这种结构的。从图中可以很显然地看出，结构改进了程序的清晰性，简化了修改及维护工作。当想修改某存贮器块内的信息构成




```
PARM—BLOCK      STRUC      结
TP—RESERVED:    DS    4    构
COMMAND:        DS    1    定
RESULT:         DS    1    义
BUFFER—START:   DS    4
BUFFER—LEN:     DS    2
PARM—BLOCK      ENDS
```

例：　不用结构元而用偏移值　　利用结构元素名

```
LPD     GA,[PP].6          LPD     GA,[PP].BUFFER—START
MOVBI   [PP].5,0           MOVBI   [PP].RESULT,0
```

图9.49　ASM-89中结构的定义及使用示例

时，只要对结构的定义做些修改就可以了，这样在程序被重新汇编时，汇编程序就能对所有的结构元引用进行适当的调整，使这些访问适应新的结构定义的要求。此外，调整结构始址寄存器的内容还可访问存贮器中的多个结构。

四、寻址方式

在前一节当中，我们已经讨论了 8089 的四种寻址方式，对应于这些寻址方式的 ASM-89 表示法如下表所示：

表 9.18 ASM-89 寻址方式表示法

表　示　法	寻　址　方　式
[ptr-reg]	基址寻址方式
[ptr-reg].offset	偏移寻址方式
[ptr-reg+IX]	变址寻址方式
[ptr-reg+IX+]	变址递增寻址方式

注：ptr-reg 是任意一个指针寄存器(GA、GB、GC 或 PP)offset 为 8 位的带符号值，它也可以是结构中的元素.

下面分别对访问简单变量、结构元和数组所应采用的存贮器寻址方式小结如下：

1．如果 ptr-reg 含有存贮器操作数的地址，那么[ptr-reg]就是指那个操作数。

2．如果 ptr-reg 含有结构的基地址，那么该结构中的元素 DATA 就可用[ptr-reg].DATA 来访问；若 DATA 距结构起点 6 个字节远，则 [ptr-reg].6 也指向同一个结构元。

3．如果 ptr-reg 含有数组的起始地址，那么利用 [ptr-reg+IX] 就可以访问数组中的元素，其中 IX 起数组"下标"的作用。例如，若 IX 的值为 4，那么[ptr-reg+IX]就指向字节数组的第五个元素或字数组的第三个元素。[ptr-reg+IX+] 与[ptr-reg+IX]所选择的元素相同，但它在得到了当前需访问元素的有效地址之后，将自动把 IX 加 1(字节操作)、加 2(字操作)或加 3(MOVP 指令)。

下面看几个例子：

```
ADDI    GA,5              寄存器,立即数
ADD     GC,[GB]           寄存器,存贮器(基址寻址)
ADDBI   [pp],10           存贮器(基址寻址),立即数
ADDB    IX,[GB].5         寄存器,存贮器(偏移寻址)
ADDB    BC,[GC].COUNT     寄存器,存贮器(偏移寻址)
ADD     [GC+IX],BC        存贮器(变址),寄存器
ADDI    [GA+IX+],5        存贮器(变址递增),立即数
ADDB    [pp].ERROR,[GA]   存贮器(偏移),存贮器(基址)
```

五、过程

ASM-89 程序可以用 CALL/LCALL 指令来调用过程(或称子程序)。CALL/LCALL 指令的第一个操作数是一个存贮器单元地址，该单元用来保存执行调用指令时 TP 的内容，即保存调用指令的下面那条指令的地址，以便过程执行完之后返回到调用点继续执行原来的程序，利用一条 MOVP 指令就能够把保护起来的指令地址装入 TP，从而过程也就返回到了 CALL/

LCALL 下面那条指令。下例说明了过程连接的一种方法。

```
        .
        .
        .
CALL-SAVE:    DS    3      (TP 的保护区)
        .
        .     建立 TP 的保护区
        .       注意：本例假定程序位于 I/O 空间。
        .             若程序在系统空间，则应使用 LPDI 指令
MOVI    GC,CALL_SAVE       (把 TP 保护区的地址装入 GC)
LCALL   [GC],DEMO          (调用过程 DEMO)
        .
        .
        .
HLT                        (程序的逻辑终点)
定义过程 DEMO
DEMO:
        .       这些是过程中的指令
        .
        .           注意：过程不能修改 GC 的内容，这是因为它指向返回地址。
        .
MOVP    TP, [GC]           (返回到发出调用的程序)。
```

当然，通道程序也可以把其参数块的前两个字用作任务指针 TP 的保存区，但是，考虑到 CPU 有可能向该通道发出“挂起”命令，所以建议不要使用这种方法。其理由就在于，挂起命令也将使当前的 TP 值保存到这两个字当中，所以这样一来就有可能复写掉了先前所保存在这两个字中的返回地址。

六、分段控制

段是一个存贮器区域，最长为 64K 字节，关于段的详细说明我们在上册当中已经给出了，故这里不再重复。对于 ASM-89 汇编程序来讲，它所产生的浮动目标模块就是一个逻辑段。ASM-89 也利用伪指令 SEGMENT 和 ENDS 来定义某个段，一般来讲，除了伪指令 END 之外，几乎所有的指令和伪指令都可以出现在 SEGMENT 和 ENDS 对当中。例：

```
CHANNEL1    SEGMENT     (段开始)
    ASM-89 的源语句
CHANNEL1    ENDS        (段结束)
            END         (汇编结束)
```

七、模块间的通讯

一个 ASM-89 模块通过使用伪指令 PUBLIC，就可以定义一些可供其它模块使用的变量(符号)。显然，这样的 PUBLIC 伪指令是必需的，因为通道程序必须使其第一条指令的地址可供启动该通道程序的 CPU 模块使用，所以至少应把指向第一条指令的符号定义为公用的(PUBLIC)。与此相反，若要使用别的模块中的公用符，则应在本模块中用伪指令 EXTRN 来定义这些符号。当把这些模块连接起来时，LINK-86 将把被说明为 PUBLIC 和 EXTRN 的那些符

号匹配起来，实现模块之间这种利用公共单元所进行的通讯。下例给出了一个 ASM-89 模块，它包括三个通道程序，其标号分别为 READ、WRITE 和 DELETE，该例主要说明 PL/M-86 程序和 ASM-86 程序是如何使用这几个“外部的”通道程序的。

例：

ASM-89 模块定义了三个公共的符号：

```
  .
  .
  .
  .
  .
  .
PUBLIC      READ, WRITE, DELETE
  .
  .
  .
  .
READ:       ;ASM89  INSTRUCTIONS FOR "READ" OPERATION
  .
  .         HLT
WRITE:      ;ASM89  INSTRUCTIONS FOR "WRITE" OPERATION
  .
  .         HLT
DELETE:     ;ASM89  INSTRUCTIONS  FOR "DELETE" OPERATION
  .
  .         HLT
```

下面是一个使用了符号“WRITE”的 PL/M-86 模块：

```
DECLARE     (READ,WRITE,DELETE)POINTER EXTERNAL;
DECLARE     PARM$BLOCK STRUCTURE
            (TP$START          POINTER,
            BUFFER$ADDR        POINTER,
            BUFFER$LEN         WORD);
  .
  .
  .
/*SET UP "WRITE" CHANNEL OPERATION*/
PARM$BLOCK,TP$START = WRITE;
```

下面这个 PL/M-86 模块使用了外部符“WRITE”。

```
EXTRN           READ.WRITE,DELETE
  .
  .
  .
  .
READ_PTR     DD    READ
WRITE_PTR    DD    WRITE
DELETE_PTR   DD    DELETE
  .
  .
  .
```

```
;PARM_BLOCK
                EVEN    ;FORCE TO EVEN ADDRESS
TP_START    DD?
BUFFER_ADDRDD?
BUFFER_LEN DW?
  .
  .
  .
;SET UP "READ" CHANNEL OPERATION
    MOV     AX,WORD PTR READ_PTR    ;1ST WORD
    MOV     WORD PTR TP_START,AX
    MOV     AX,WORD PTR READ_PTR    ;2ND WORD
    MOV     WORD PTR TP_START,AX
```

ASM-89 模块也可以使用外部定义的符号，但它规定外部符号只能作为伪指令 DD 的操作数。例：

假设 PL/M-86 程序说明了公共符号"BUFFER"：

```
  .
  .
  .
DECLARE     BUFFER(80) BYTE PUBLIC;
  .
  .
  .
```

下面这段 ASM-89 程序获得了公共符号"BUFFER"的地址：

```
  .
  .
  .
EXTRN     BUFFER
  .
  .
  .
BUF_ADDRESS DD BUFFER
  .
  .
  .
LPD     GA,BUF_ADDRESS     (指向系统空间的缓冲区)
  .
  .
  .
```

八、程序设计举例

作为 ASM-89 的小结，我们来设计一个 ASM-89 程序段，让它从软盘驱动器中读出一个物理记录(段)。与 8089 相连的是一个 8271 软盘控制器，由它来控制软盘驱动器。8271 常驻在 8089 的 I/O 空间当中。假设硬件地址译码如下所示：

1．读单元 FF00H 选择的是 8271 状态寄存器。

2．写单元 FF00H 选择的是 8271 命令寄存器。

3．读单元 FF01H 选择的是 8271 结果寄存器。

4．写单元 FF01H 选择的是 8271 参数寄存器。

5．单元 FF04H 提供 8271 的 DMA 响应(DACK)信号。

在本程序中，参数块和 8271 寄存器都被构造成结构，用寻址结构元的方法来访问参数及 8271 的寄存器。指针寄存器 PP 指向参数块的始地址，GC 中的内容为 FF00H，让它指向 8271 的寄存器。为了使 CPU 能够启动该程序，故把程序的入口标号 START 定义为是一个 PUBLIC 符号。寄存器 IX 被用来控制重复传送的次数，即如果传送不成功(8271 结果寄存器的第四位不为 0)，那么程序就重新开始又一次传送过程，为了防止无休止地重复下去，故用 IX 作为重复传送次数的控制工具，保证程序最多重传 10 次。由于 8271 一旦收到最后一个参数，就将自动请求 DMA 传送，所以最后一个参数(扇区号)必须紧接在 XFER 指令之后发给 8271。

下面给出该 ASM-89 程序的简单逻辑框图。

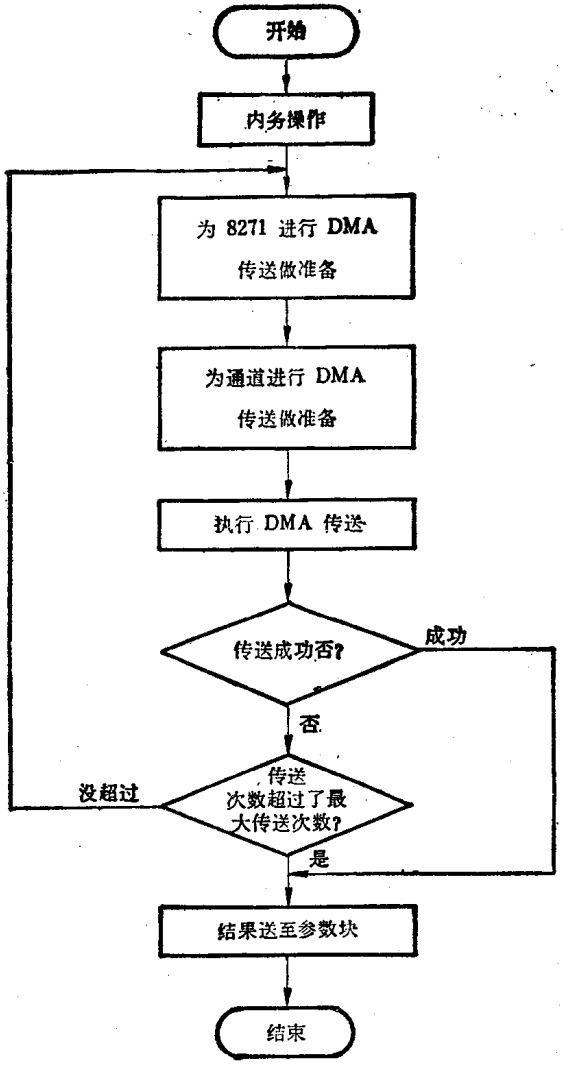


图 9.50　读软盘程序框图

对应的程序编码如下：

```
8089 ASSEMBLER
ISIS-II 8089 ASSEMBLER V1.0 ASSEMBLY OF MODULE FLOPPY
OBJECT MODULE PLACED IN :F0:FLOPPY.OBJ
ASSEMBLER INVOKED BY ASM89 FLOPPY.A89
                          1
0000                      2  FLOPPY     SEGMENT
                          3  ;***
                          4  ;*** 8089 PROGRAM TO READ SECTOR FROM
                              FLOPPY DISK
                          5  ;***
                          6
                          7  ;*** LAY OUT PARAMETER BLOCK(定义参数块的格
                              式)
                          8  PARM_BLOCK     STRUC
0000                      9    RESERVED_TP;     DS  4
0004                     10    BUFF_PTR;        DS  4
0008                     11    TRACK;           DS  1
0009                     12    SECTOR;          DS  1
000A                     13    RETURN_CODE;     DS  1
000B                     14    PARM_BLOCK     ENDS
                         15
                         16  ;*** LAY OUT 8271 DEVICE REGISTERS. (8271 设备
                              寄存器组成的结构定义)
                         17  FLOPPY_REGS     STRUC
0000                     18    COMMAND_STAT;    DS  1
0001                     19    PARM_RESULT;     DS  1
0002                     20    FLOPPY_REGS ENDS
                         21
                         22  ;***8271 ADDRESSES.
  FF00                   23  FLOPPY_REG_ADDR     EQU     0FF00H
                             ;LOW_ADDRESSED REGISTER
  FF04                   24  DACK_8271          EQU     0FF04H
                             ;DMA ACKNOWLEDGE
                         25
                         26  ;*** MAKE PROGRAM ENTRY POINT ADDRESS (让
                              其它模块可以使用程序的入口地址)
                         27  ;    AVAILABLE TO OTHER MODULES.
                         28  PUBLIC     START
```

```
                    29
                    30  ;*** CLEAR RETURN CODE IN PARAMETER BLOCK.
                        (参数块中的返回码清 0)
0000 0A4F 0A 00     31  START:    MOVBI [PP]. RETURN__CODE,0
                    32
                    33  ;***INITIALIZE RETRY COUNT. (设置重传的最大次
                        数为 10)
0004 B130 0A00      34        MOVI    IX, 10
                    35
                    36  ;***POINT GC AT LOW__ORDER 8271 REGISTER.
0008 5130 00FF      37        MOVI    GC, FLOPPY__REG__ADDR
                    38
                    39  ;***SEND COMMAND SEQUENCE TO 8271,HOLDING
                        FINAL PARM. (等到 8271 不忙时,给 8271 发命令)
                    40  ;***WAIT UNTIL 8271 IS NOT BUSY.
000C EABA 00 FC     41  RETRY:    JNBT    [GC]. COMMAND__STAT, 7,
                                          RETRY
                    42  ;***SEND "READ SECTOR,DRIVE 0" COMMAND.
                        (从 0 号驱动器读命令)
0010 0A4E 00 12     43            MOVBI   [GC]. COMMAND__STAT, 012H
                    44  ;***SEND TRACK ADDRESS PARAMETER. (道号送
                        到参数块)
0014 0293 08 02CE 01 45   MOVB    [GC]. PARM__RESULT,[PP]. TRACK
                    46
                    47  ;***LOAD CHANNEL CONTROL REGISTER SPECIFY-
                        ING: (设置通道控制寄存器: I/O→存贮器,源同步,GA
                        指向源,外部结束位移 = 0)
                    48  ;     FROM PORT TO MEMORY,
                    49  ;     SYNCHRONIZE ON SOURCE,
                    50  ;     GA POINTS TO SOURCE,
                    51  ;     TERMINATE ON EXT,
                    52  ;     TERMINATION OFFSET = 0.
001A D130 2088      53            MOVI   CC,08820H
                    54
                    55  ;***SET SOURCE BUS = 8,DEST BUS = 16. (设置逻辑
                        总线宽度)
001E A000           56            WID    8,16
                    57
                    58  ;***POINT GB AT DESTINATION, GA AT SOURCE.
```

```
0020 238B 04          59            LPD     GB,[PP]. BUFF PTR
0023 1130 04FF        60            MOVI    GA,DACK_8271
                      61
                      62  ;***INSURE THAT 8271 IS READY FOR LAST PARA=
                          METER.(保证在发出最后一个参数之前 8271 准备好)
0027 AABA 00 FC       63  WAIT1:    JNBT    [GC]. COMMAND_STAT, 5,
                                            WAIT1
                      64
                      65  ;***PREPARE FOR DMA.
002B 6000             66            XFER
                      67
                      68  ;***START DMA BY SENDING FINAL PARAMETER
                          TO 8271.(给 8271 最后一个参数,开始 DMA)
002D 0293 09 02CE 01  69            MOVB    [GC]. PARM_RESULT, [PP].
                                            SECTOR
                      70
                      71  ;***PROGRAM RESUMES HERE FOLLOWING EXT.
                      72
                      73  ;***IF TRANSFER IS OK THEN EXIT,ELSE TRY AG-
                          AIN.(若传送正确,转 EXIT)
0033 6ABE 01 05       74            JBT     [GC]. PARM_RESULT,3,EXIT
                      75
                      76  ;***DECREMENT RETRY COUNT.
0037 A03C             77            DEC     IX
                      78
                      79  ;***TRY AGAIN IF COUNT NOT EXHAUSTED.(若
                          传送不正确且还没传送到 10 次,则再次传送)
0039 A840 D0          80            JNZ     IX,RETRY
                      81
                      82  ;***WAIT UNTIL 8271 IS NOT BUSY.(直等到 8271
                          不忙)
003C EABA 00 FC       83  EXIT:     JNBT    [GC). COMMAND_STAT,7,EXIT
                      84
                      85  ;***SEND"READ RESULT"COMMAND TO 8271.(发
                          "读结果"命令给 8271)
0040 0A4E 00 2C       86            MOVBI   [GC]. COMMAND_STAT, 02CH
                      87
                      88  ;***WAIT FOR RESULT.
0044 8ABA 00 FC       89  WAIT2:    JNBT.   [GC]. COMMAND_STAT, 4:
```

```
                                                    WAIT2
                                90
                                91  ;***POST RESULT IN PARAMETER BLOCK FOR
                                    CPU.(结果送到参数块)
0048 0292 01 02CF OA            92          MOVB    [PP]. RETURN__CODE,[GC].
                                                    PARM__RESULT
                                93
                                94  ;***INTERRUPT CPU.(中断 CPU)
004E 4000                       95          SINTR
                                96
                                97  ;***STOP EXECUTION.
0050 2048                       98          HLT
                                99
0052                           100  FLOPPY  ENDS
                               101          END
SYMBOL TABLE
DEFN VALUE TYPE NAME
  10    0004   SYM   BUFF__PTR
  18    0000   SYM   COMMAND__STAT
  24    FF04   SYM   DACK__8271
  83    003C   SYM   EXIT
   2    0000   SYM   FLOPPY
  17    0000   STR   FLOPPY__REGS
  23    FF00   SYM   FLOPPY__REG__ADDR
   8    0000   STR   PARM__BLOCK
  19    0001   SYM   PARM__RESULT
   9    0000   SYM   RESERVED__TP
  41    000C   SYM   RETRY
  13    000A   SYM   RETURN__CODE
  12    0009   SYM   SECTOR
  31    0000   PUB   START
  11    0008   SYM   TRACK
  63    0027   SYM   WAIT1
  89    0044   SYM   WAIT2
ASSEMBLY COMPLETE WO ERRORS FOUND
```

§9.8.2 iAPX86/11、iAPX88/11 的程序设计及实例

8086 CPU 与 8089 I/O 处理器结合起来就组成了 iAPX 86/11 系统，类似地，若以 8088 为

CPU 则得到 iAPX88/11，在上册当中我们已经介绍了 8086/8088 的汇编语言 ASM-86 及高级语言 PL/M-86，前面我们又简单地介绍了 8089 的汇编语言 ASM-89，那么从整个系统的角度来讲，怎样才能把这些用不同语言编写的程序模块结合成一个整体呢？其实，读者不必为此犯愁，这一任务是由连接和定位程序 LINK-86 完成的，它能把多个浮动的目标程序模块结合成一个单一的浮动模块，其输入模块可用 ASM-89、ASM-86、PL/M-86 语言编写。实际上 LINK-86 的主要功能就是满足各模块的外部引用请求，所谓外部引用也就是指 ASM-86 或 ASM-89 中用伪指令 EXTRN 定义的那些符号，和在 PL/M-86 中被说明为 EXTERNAL 的任一符号。每当 LINK-86 遇到外部引用时，它就搜索其它模块，寻找名字相同且被说明成 PUBLIC 的符号，若找到了这一符号，则说明该外部引用是合法的，故用该符号的目标地址取代对应的外部引用。

在 iAPX86/11、88/11 当中，最一般的外部引用是通道程序的地址。由于通道程序是单独汇编的，所以处理 CPU 程序的翻译程序一般也就不知道通道程序的地址，为了使 CPU 程序能够启动通道程序，它就必须把通道程序的地址定义为外部符号，这样就可以由 LINK-86 从 ASM-89 通道程序得到这个地址（放在参数块的头两个字当中），从而起到了把两个程序模块连接起来的作用，当然，ASM-89 必须用伪指令 PUBLIC 来定义该符号。

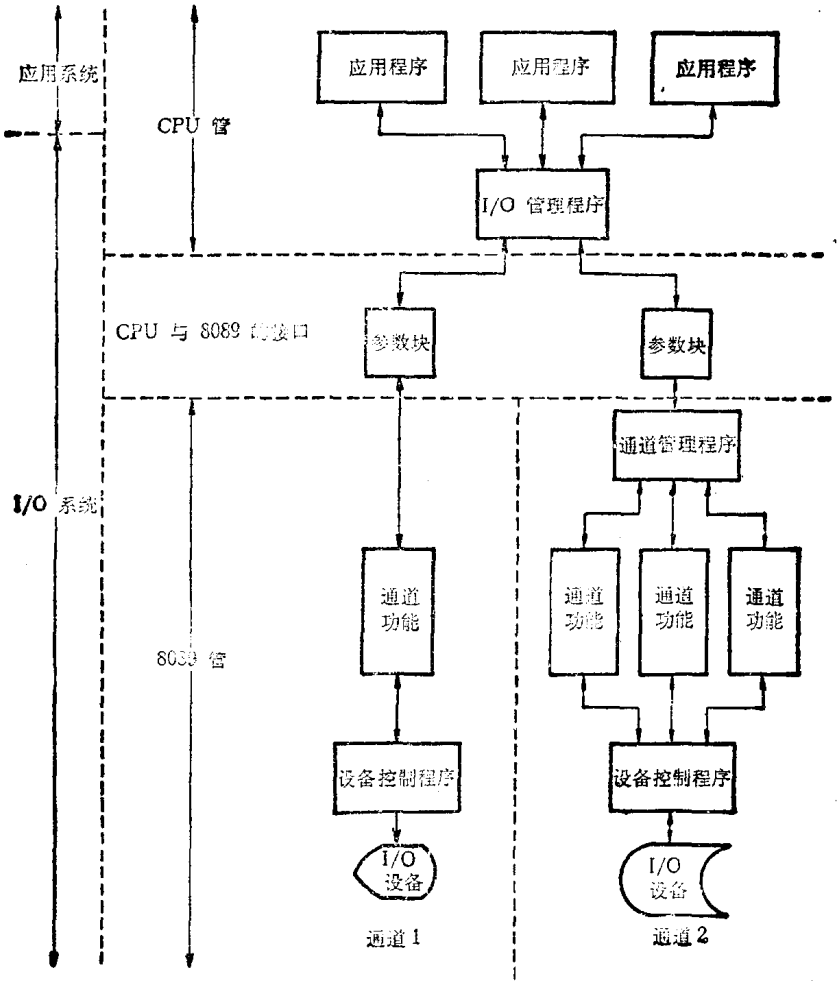


图 9.51 I/O 系统的设计

下面简单地讲一讲利用 8089 的输入/输出系统的设计方法。一般来讲，我们提倡分层地设计 I/O 系统，让应用程序仅仅看到系统的最顶层，至于 I/O 设备的物理特性和操作细节则由较低层处理。图 9.51 表示在用了 8089 的系统当中是怎样应用这种设计方法的，同样，类似的方法也可以推广到具有多个 8089 的大型系统中。

利用这种设计方法所得到的结果就是：应用系统与 I/O 系统是分开的，应用程序本身不执行 I/O 操作，而只是向 I/O 管理程序发出 I/O 请求。在面向文件的 I/O 系统当中，应用程序向文件系统发出请求，然后再由文件系统去调用 I/O 管理程序。这样的 I/O 请求是以数据的逻辑结构表达的，即访问一个记录、某个行、某个信息等等，而不需要使用任何与设备有关的信息，例如设备地址、扇区号等信息。

I/O 管理程序把应用程序的 I/O 请求变换为一个参数块，并调度某个通道程序来完成该操作。由于各通道都是由 I/O 管理程序所控制的，所以管理程序应该知道通道和 I/O 设备之间、通道控制块 CB 与通道程序之间的对应关系，也应该知道所有参数块的格式，此外，它还要协调、监督 BUSY 标志，响应通道产生的中断请求之类的通道事件。

CPU 的 I/O 管理程序和 8089 的通道程序之间的软件接口是在参数块中被定义的，例如，I/O 管理程序若要传送通道程序需用的系统存贮器地址，就应当通过参数块 PB。这种把整个接口都归纳到一起的方法使得系统比较易于理解，修改起来比较方便，并能减小由于修改而产生的副作用。这种设计方法也可以推广到其它应用系统中去。

在图 9.51 所示的系统当中，通道 1 上运行的是一个较简单的通道程序，它只具有一种功能，故被设计成一个简单程序；但通道 2 上运行的通道程序就比较复杂了，它执行三种功能，如读、写、删除三种操作，这几个功能是分别构造出来的。当然，这些功能也可以用过程来实现，然后，由通道管理程序根据参数块中的内容确定应该调用的是哪个过程。

最后，我们来看几个程序例子。第一个例子说明 CPU 是怎样初始化一组 8089 的，以及怎样调度 8089 的两个通道程序进行工作的。该程序用 PL/M-86 语言编写，它对两个 8089 进行初始化处理，但是，初始化任意数目的 8089 可以采用同样的方法。假设程序的运行环境是这样的：

1. 8086 CPU(16 位的系统总线)；
2. 两个远程方式的 8089，它们以请求/批准模式 1 共用 8 位的本地 I/O 总线；
3. CPU I/O 空间的四个通道地址译码产生 8089 的通道注意信号；
4. 通道程序驻留在 8089 的 I/O 空间当中；
5. 用一个 8089 去控制一个 CRT 终端及一些扫描键盘，另一个 8089 的功能在此没作规定。

本程序把系统结构块 SCB、系统结构指示器 SCP 和 CB（通道控制块）都说明为结构，由于有两个 8089，而每个 8089 都有自己的通道控制块，所以给出了两个 CB 说明。更进一步讲，每个 8089 又都有两个通道，因而把 CB 说明成二元的结构数组，即数组的元素为结构，这些数组元分别对各自对应的通道信息进行定义。在这样一些结构当中，除了 SCP 被定位在固定的系统单元 FFFF6H 以外，其它结构都无需定位在特定的地址处，它们是通过一串指示器连到一起的。

为了在 PL/M-86 程序和 CRT 之间进行信息传输，程序中还定义了两个颇简单的参数块。每个参数块都包含一个指向信息区起点的指示器，还包含一个信息长度计数器。程序每次初

始化一个 8089，为了初始化第一个 8089(8089 A)，首先填好 SYSBUS 和 SOC 字段，并将这些结构块连接起来，然后把通道 1 的 BUSY 标志置为 FFH，这样，8089 就可以通过监视该标志来确定初始化是否已经完成，即当该标志为 0H 时，说明初始化已告完成。利用 PL/M-86 的 OUT 语句，向通道 1 发出通道注意信号，这就使得 8089 进入等待状态，直到 8089 清除了通道 1 的 BUSY 标志为止，此时结束该 8089 的初始化过程，至于 OUT 所送出的数据 0H，它没有什么作用。

完全类似地可以初始化第二个 8089(8089B)，首先要改变 SCB 中的指示器(CB$PTR)，让其指向 8089B 的通道控制块 CB，有必要的话还要更改 SOC 字段的内容。由于 8089B 是作为从设备出现的，所以应该向通道 2 发出通道注意信号来开始初始化过程，这是与初始化 8089A 时所不同的。

在两个 8089 都准备好了之后，就开始调度通道程序，例中我们没有给出通道程序的代码。实际上所做的工作就是把通道程序(任务块)连到参数块上，把参数块连到通道控制块上，再将信息地址及长度填入参数块，设置通道命令字 CCW，最后发出适当的通道注意信号，这就使得相应的通道开始执行起来了。

下面就给出这段程序的 PL/M-86 编码：

```
/*ISSUE CA FOR CHANNEL1,INDICATING IOP IS MASTER*/      (给通道1发CA信号,表示8089A是主设备)
OUT(IOP$A$CH1)=01H;
/*WAIT UNTIL FINISHED*/                                 (等待,直到初始化8089A工作完成)
DOWHILE CB$A(0).BUSY=CHANNEL$BUSY;
   END;
/*PREPARE CONTROL BLOCKS FORIOP$B*/                     (为8089B准备通道控制块)
SCB,CB$PTR=@CB$B(0);
CB$B(0),BUSY=CHANNEL$BUSY;
CB$B(1).BUSY=CHANNEL$CLEAR;
/*ISSUE CA FOR CHANNEL2,INDICATING SLAVE STATUS*/       (向通道2发CA,表示8089B为从设备)
OUT(IOP$B$CH2)=0H;
/*WAIT UNTIL IOP IS READY*/                             (等待,直到8089已准备好)
DO WHILE CB$B(0).BUSY=CHANNEL$BUSY;
   END;
/*
*SEND SIGN ON MESSAGE TO CRT CONTROLLED                (发送信息到8089A的通道1所控制的
*BY CHANNEL 1 OF IOP$A                                  CRT)
*/
/*WAIT UNTIL CHANNEL IS CLEAR, THEN SET TO BUSY*/       (等到8089的通道1空闲,然后设置忙标志)
DO WHILE LOCKSET(@CB$A(0),BUSY,CHANNEL$BUSY);
   END;
/*SET CCW AS FOLLOWS:                                   (CCW设置如下:优先权=1,无总线加载限制,禁止中断,开始执行I/O空间的程序)
*    PRIORITY=1,
*    NO BUSLOAD LIMIT,
*    DISABLE INTERRUPTS,
*    START CHANNEL PROGRAM IN I/O SPACE*/
CB$A(0).CCW=10011001B;
```

```
/*LINK MESSAGE PARAMETER BLOCK TO CB*/                    (把信息参数块连到 CB)
CB$A(0).PB$PTR=@MESSAGE$PB;
/*FILL IN PARAMETER BLOCK*/                               (填入参数块)
MESSAGE$PB.TB$PTR=DISPLAY$TB;
MESSAGE$PB.MSG$PTR=@SIGN$ON;
MESSAGE$PB.MSB$LENGTH=LENGTH(SIGN$ON);
/*DISPATCH THE CHANNEL*/                                  (调度该通道,让其开始执行)
OUT(IOP$A$CH1)=0H;
/*
*DISPATCH CHANNEL 2 OF IOP$A TO                          (调度 8089A的通道2,让它不断扫描键盘,
                                                          在一个完整的信息准备好了之后,中断)
*CONTINUOUSLY SCANKEYBOARD,INTERRUPTING
*WHEN A COMPLETE MESSAGE IS READY
*/
/*WAIT UNTIL CHANNEL IS CLEAR,THEN SET TO BUSY*/          (等待,直到 8089A 的通道 2 准备好,然后
                                                          置忙标志)
DO WHILELOCKSET @CB$A(1).BUSY,CHANNEL$BUSY);
   END;
/*SET CCW AS FOLLOWS,                                     (CCW 的设置情况如下:优先权位=0,总
                                                          线加载限制,允许中断,开始执行 I/O 空
                                                          间的通道程序)
*    PRIORITY=0
*    BUS LOAD LIMIT,
*    ENABLE INTERRUPTS,
*    START CHANNEL PROGRAM IN I/O SPACE*/
CB$A(1).CCW=00110001B;
/*LINK KEYBOARD PARAMETER BLOCK TO CB*/                   (把键盘的参数块连到 CB)
CB$A(1).PB$PTR=@KEYBD$PB;
/*FILL IN PARAMETERBLOCK*/                                (填入参数块)
KEYBD$PB.TB$PTR=KEYBD$TB;
KEYBD$PB.BUFF$PTR=@KEYBD$BUFF;
KEYBD$PB.MSG$SIZE=0H;
/*DISPATCH THE CHANNEL*/                                  (调度通道 2,让其开始执行)
OUT(IOP$A$CH2)=0H;
/*ASSIGN NAMES TO CONSTANTS*/                             (用一些名字来表示常数)
DECLARE      CHANNEL$BUSY      LITERALLY'0FFH';
DECLARE      CHANNEL$CLEAR     LITERALLY'0H';
DECLARE      CR/*CARR.RET.*/   LITERALLY'0DH';           (回车)
DECLARE      LF/*LINEFEED*/    LITERALLY'0AH';           (换行)
DECLARE      DISPLAY$TB        LITERALLY'200H';
DECLARE      KEYBD$TB          LITERALLY'600H';
DECLARE  /*IOP CHANNEL ATTENTION ADDRESSES*/             (产生通道注意信号的四个地址)
IOP$A$CH1    LITERALLY         '0FFE0H',
IOP$A$CH2    LITERALLY         '0FFE1H',
IOP$B$CH1    LITERALLY         '0FFE2H',
IOP$B$CH2    LITERALLY         '0FFE3H';
DECLARE  /*CHANNEL CONTROL BLOCK FOR IOP$A               (8089A 的通道控制块)
             CB$A(2)           STRUCTURE
             (BUSY             BYTE,
             CCW               BYTE,
             PB$PTR            POINTER,
             RESERVED          WORD);
```

```
DECLARE  /*CHANNEL CONTROL BLOCK FOR IOP$B*/          (8089B 的通道控制块)
           CB$B(2)          STRUCTURE
           (BUSY            BYTE,
           CCW              BYTE,
           PB$PTR           POINTER,
           RESERVED         WORD);
DECLARE  /*SYSTEM CONFIGURATION BLOCK*/              (系统结构块)
           SCB              STRUCTURE
           (SOC             BYTE,
           RESERVED         BYTE,
           CB$PTR           POINTER);
DECLARE  /*SYSTEM CONFIGURATION POINTER*/           (系统结构指针)
           SCP              STRUCTURE
           (SYSBUS          BYTE,
           SCB$PTR          POINTER)AT(0FFFF6H);
DECLARE  MESSAGE$PB STRUCTURE
           (TE$PTR          POINTER,
           MSG$PTR          POINTER,
           MSG$LENGTH       WORD);
DECLARE  KEYBD$PB STRUCTURE
           (TP$PTR          POINTER,
           BUFF—PTR         POINTER,
           MSG$SIZE         WORD);
DECLARE  SIGN$ON BYTE(*)DATA
           (CR,LF,'PLEASE ENTER USER ID');
DECLARE  KEYB$DBUFF BYTE(256);
/*
*INITIALIZEIOP$A,THENIOP$B                          (先初始化 8089A,再初始化 8089B)
*/
/*PREPARE CONTROL BLOCKS FOR IOP$A*/
SCP.SCB$PTR=@SCB;
SCP.SYSBUS=01H;/*16-BIT SYSTEM BUS*/                (16 位系统总线)
SCB.SOC=02H;/*RQ/GT MODE1,8-BIT I/O BUS*/           (RQ/GT 模式 1, 8 位 I/O 总线)
SCB.CB$PTR=@CB$A(0);
CB$A(0).BUSY=CHANNEL$BUSY
CB$A(1).BUSY=CHANNEL$CLEAR;
```

图 9.52　8089 的初始化及通道程序的调度

接下来的两个例子是用 ASM-89 写出的，主要目的是说明 8089 的指令系统及常用的一些寻址方式。一个例子完成在存贮器到存贮器的 DMA 传送，另一个例子完成寄存器的保护功能。

图 9.53 给出了一个通道程序，它只用了七条指令就能完成存贮器到存贮器的数据传块送，该程序可在系统存贮器的两个存贮空间之间传送多达 64K 字节的数据。假设系统总线的宽度为 16 位，且设 CPU 通过检查通道的 BUSY 标志来确定程序是否已经完成。

为了获得最大的传送速度，程序在每个 DMA 传送周期都将封锁总线，这就能够保证在 DMA 的存和取操作间隔内，其它处理器不会获得总线。若把该通道的 CCW 中的优先权位置为 1 而把其它通道的相应位置为 0，则又可以有效地防止传送期间其它通道的运行。传送结束条件只有一个——字节数，故当且仅当由 CPU 所规定传送的字节数传送完了之后，才停止传送过程。DMA 传送是在执行完 WID 指令时开始的，一旦传送结束，就执行 HLT 指令。

```
MEMEXAMP        SEGMENT
;**MEMORY TO-MEMORY TRANSFER PROGRAM.** (存贮器到存贮器的传送程序)
PB              STRUC
TP—RESERVED:    DS    4
FROM—ADDR:      DS    4
TO—ADDR:        DS    4
SIZE:           DS    2
PB              ENDS
;POINT GA AT SOURCE,GB AT DESTINATION. (GA 为源指针,GB 为目标指针)
                LPD     GA, [PP].FROM—ADDR
                LPD     GB, [PP],TO—ADDR
;LOAD BYTE COUNT INTO BC (需传送的字节数送到 BC)
                MOV     BC,[PP].SIZE
;LOAD CC SPECIFYING: (设置通道控制寄存器 CC 如下: 存贮器到存贮器,不翻译,不同步, GA 指向源,传送期
                        间加封锁,不加链,BC=0 时结束且位移量=0)
;   MEMORY TO MEMORY,
;   NOTRANSLATE,
;   UNSYNCHRONIZED,
;   GA POINTS TO SOURCE,
;   LOCK BUS DURING TRANSFER,
;   NO CHAINING,
;   TERMINATING ON BYTE COUNT, OFFSET=0.
                MOV     CC,0C208H
;PREPARE CHANNEL FOR TRANSFER. (令通道准备传送)
                XFER
;SET LOGICAL BUS WIDTH. (设置逻辑总线宽度)
                WID     16,16
;STOP EXECUTION AFTER DMA. (DMA 传送结束后,停止执行)
                HLT
MEMEXAMP        ENDS
                END
```

图 9.53 存贮器之间的数据传送

我们已经知道，CPU 程序的一条"挂起"通道命令可以中断一个通道程序的执行，通道一旦接到挂起命令，就将把任务指示器和 PSW 保存到参数块的前两个字中．通过发出"恢复"命令，又可以从保护区取回 TP 和 PSW，重新恢复执行被挂起的程序．

倘若在挂起和恢复操作之间，CPU 又要求通道执行另一个通道程序，由于该通道程序可能要用到一些寄存器，而这些寄存器是被挂起的程序已经用过且还将要用到的，所以为了能够使被挂起的程序在后面某个时候恢复执行，必须将寄存器的原始内容保护起来，并在恢复程序之前恢复这些寄存器的内容。

图 9.54 是完成寄存器保护功能的一段 ASM-89 程序，它保护的只是一些通道寄存器．因为只有 PP 指向的存贮区域可作此用途，故把这些寄存器都保存在参数块中．类似地，可以写出恢复这些寄存器的程序。

```
SAVEREGS        SEGMENT
;SAVE ANOTHER CHANNEL'S REGISTERS IN PB (把另一通道的寄存器保存到参数块 PB 当中)
PB              STRUC
TP—RESERVED:    DS      4
GA—SAVE:        DS      3
GB—SAVE:        DS      3
```

```
GC—SAVE:    DS      3
IX—SAVE:    DS      2
BC—SAVE:    DS      2
MC—SAVE:    DS      2
CC—SAVE:    DS      2
PB          ENDS
            MOVP    [PP].GA—SAVE,GA
            MOVP    [PP].GB—SAVE,GB
            MOVP    [PP].GC—SAVE,GC
            MOV     [PP].IX—SAVE,IX
            MOV     [PP].BC—SAVE,BC
            MOV     [PP].MC—SAVE,MC
            MOV     [PP].CC—SAVE,CC
            HLT
SAVEREGS    ENDS
            END
```

图 9.54 寄存器保护程序

思 考 题

9.1 简述计算机输入/输出系统的发展过程，输入/输出系统发展的动力及基础是什么？

9.2 什么是 DMA 传送方式？什么是通道？它们之间有什么相似之处，又有什么区别？

9.3 I/O 处理机有什么优点？8089 I/O 处理器又有些什么特色？

9.4 什么是通道命令？8089 是如何获得、执行通道命令的？

9.5 作为一个专用的 I/O 处理器，8089 与其 CPU 之间是如何进行通信的？8089 与 CPU 之间的通信机构是怎样的，它有些什么优点？

9.6 给出 8089 执行一次 DMA 传送的全过程。

9.7 8089 具有哪两种工作方式？它们各有什么特点？

9.8 8089 可分成哪几个功能部件，它们各起什么作用？

9.9 试述 8089 中通道的概念、结构，8089 的两个通道是如何工作的？通道寄存器有些什么特点？

9.10 8089 的 DMA 传送可由哪几种方式终止？为什么设置这许多终止条件，试分别说出理由。

9.11 8089 是如何与其它处理器共享总线的？

9.12 参见图 9.54，用 ASM-89 编写一段程序，从某存贮器区域中恢复寄存器的内容。

第十章　iAPX 86/20,88/20 数值处理机的硬件与软件

§10.1　概　　述

数值数据处理机(Numeric Data Processor)iAPX 86/20 和iAPX 88/20 主要是为了提高数值运算的速度而设计的一种双片处理机。它在指令这一级提供了高精度的整数和浮点运算能力，其数值运算既包含加、减、乘、除这样一些普通运算，也包括一般通用机所不具备的平方根、乘幂、对数以及三角函数运算。正因为如此，我们把它称为数值数据处理机，简称为数值处理机。

在科学研究、工程计算、航海航空及国防等很多领域中，经常需要对大量的数据进行实时处理，并要求计算三角函数、对数函数、指数函数这样一些"超越"函数的值，在以往的微型计算机甚至一般的中小型乃至大型计算机当中，这些函数值大多是借助于软件来计算的，即靠编一些子程序来计算这些函数的值，也就是说，每计算一次函数值就要调用一段子程序，当然，在某些应用中，这种调用请求不是很频繁，故这种方案从某种意义上说还是可以接受的。但是也应当看到，当经常需要计算这些函数值时，由于子程序的频繁调用，就会显著降低机器处理问题的速度。很显然，这样的处理方法不适宜用于复杂的数据处理系统。

为了提高计算机的数值运算能力，就必须提供相应的硬件支持，当然这会增加机器设计的难度，从经济上讲也提高了机器的价格，这些又是我们所一直设法避免的。实际上，从前面的介绍当中我们也可以看出，对一些不太复杂的应用来讲，并不需要很强的数值处理能力，因此，考虑到价格因素，一般的通用数据处理机都不具有这些硬件特色。只是在必要时，选择一个容易使用的、具有高级硬件软件支持的特殊处理器来满足这种特殊需要，这显然是解决问题的一个有效办法。除了数值运算之外，类似的处理器还可以专用于数据库管理、通讯系统以及 I/O 管理，总之，从计算机结构发展的趋势来看，一个很有前途的技术方向就是制造一些专用的处理器，根据应用的需要把它们加到原有的通用计算机上去，然后就可进行负载分担，让它们分别并行地执行各自的任务，从而提高整个系统的性能。

Intel 公司的 8087 就是一种专用于从事数值运算的处理器，它能实现多种类型的数值操作，也可以进行一些超越函数(如三角函数、对数函数)的计算，即在 8087 当中，这些函数值的计算被硬化了，也就是说不再使用软件而是靠硬件来计算这些函数，从而大幅度地提高了微型机的数值运算速度。

在前一章中我们已经讲过，微型计算机的发展可分为三代，第一代和第二代微型机系统的性能主要受到以下三个方面的限制：存贮容量、I/O 处理的速度以及数值运算。第三代的 8086/8088 打破了 64K 字节内存的框框，直接寻址范围达 1 兆字节，从而能够承担更大的任务；8089 I/O 处理器又解决了许多 I/O 处理方面的问题，使得微型机能够有效地应用于 I/O 频繁的设计当中；为了解决第三个问题，也为了提高 8086/8088 系列的性能与竞争能力，推出了数值协处理器 8087(NPX)，这就使得微型机能够在对计算有很高要求的应用当中大显身手。

Intel 公司在不断地改进数值运算工具的努力中，数值协处理器 8087 是其较为先进的发展成果，图 10.1 说明了 Intel 公司的数值运算产品的更新发展过程。

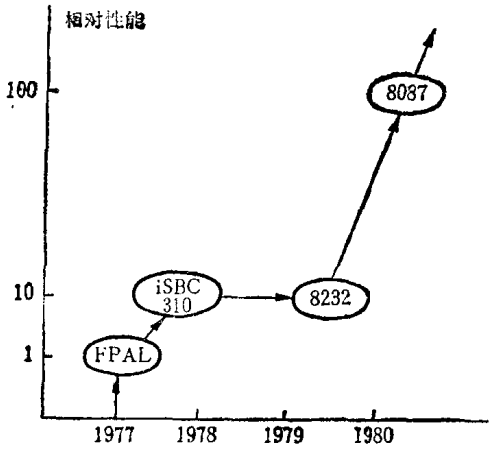


图 10.1 8087 的形成过程

早在七十年代初期，Intel 公司就作出许诺：把微型机的计算能力从整数的加减扩大到对实数进行处理。1977 年，该公司采纳了表示实数的浮点格式标准，其浮点运算库 FPAL 就是利用这一标准格式的第一个产品，实际上，FPAL 是 8080/8085 的一组子程序，它们用单精度的实数(32 位)实现了算术运算以及有限的几个标准函数的计算，对于 1.6 MHz 的 8080A CPU 来讲，执行一次 FPAL 的乘法大约需要 1.5ms；高速运算部件 iSBC 310^{TM} 实际上是实现 FPAL 功能的一块 iSBC^{TM} 单板机，它把执行一次单精度乘法所花的时间缩短到 100μs 左右；8232 是 8080/8085 系列的单片算术处理器，它实现一次单精度的乘法大约也需要 100μs，此外，8232 还可以接受双精度(64 位)的操作数，一次双精度的乘法运算约需 87.5μs；1979 年，电气与电子工程师协会 IEEE 为小型机及微型机的浮点运算推荐了一个工业标准，它的宗旨是为了便于计算机之间数值处理程序的移植，并为开发正确可靠的软件提供一个一致的编程环境，8087 就是在此环境中，对单精度及双精度等等类型的数据进行运算的单片硬件，由于前面几种产品所采用的数据格式也符合 IEEE 标准，因而 8087 也就同 Intel 公司的早期数值处理产品兼容，而且为 8087 所编制的程序也一定可以用于符合 IEEE 浮点数标准的未来产品之中。从图 10.1 中可以看出，8087 的执行速度大约是 8232 的 10 倍，是 FPAL 的 100 倍，它执行一次单精度实数乘法所花的时间约为 19μs，进行一次双精度乘法约需 27μs。

数值处理机 iAPX 86/20、88/20 分别是由数值协处理器 8087(NPX)与 8086 CPU 或 8088 CPU 一起组成的，它们将 8086、8088 的数值处理能力提高了将近 100 倍。iAPX 86/20、88/20 在原来 iAPX 86/10、88/10 的基础上又增加了几十条指令及 8 个 80 位宽的寄存器，这些也是提高数值运算速度所必不可少的物质基础。8087 能够在不带舍入误差的前题下处理 18 位的 2—10 进制数(BCD)，所处理的整数长度可达 64 位($\pm 10^{18}$)，因而在事务处理和商业部门也大有其用武之地。8087 在财会方面、科学计算以及工程设计方面的适用性，进一步显示出其比“浮点运算部件”更为优越，浮点运算部件目前仍然用在包括中小型机以及大型机在内的许多计算机系统之中。作为工作在最大模式的 8086/8088 的一个辅助处理器，8087 有效地扩充了 CPU 的寄存器以及指令组，并且增加了几种新的数据类型。一般来说，程序员不会感到 8087 是一个分开的单独设备，而只是感到 CPU 的计算能力有了极大的提高。

数值处理机(NDP)的基础部分是一个 8086 或 8088 微处理器。8086 与 8088 都是通用的微处理器，它们适用于一般的数据处理过程，对于这样一些过程来讲，一般只要求机器能够快速有效地传输数据和快速有效的程序控制指令，而实际的数值运算却是比较简单的，8086 及 8088 都能经济有效地满足这些要求。但是，正象前面所指出的那样，有些应用往往需要更为有力的数值运算指令及数据类型，现实中常要求进行浮点运算，要求计算平方根、三角函数、对数函数的值，这时仅仅提供整数类型及其对应的加、减、乘、除运算能力已不能满足精确、高

速、易用的需要。这些功能的实现并不简单，需要增加不少费用。由于在一些不太复杂的应用当中并不需要这种性能，考虑到价格因素，因而一般的通用数据处理机不具备这些特色，只是在需要时，选择一个容易使用的、具有高级硬件软件支持的专用数值运算处理器来满足实际应用的要求。

数值处理机（NDP）具有这些特征，它提供了所需要的数据类型与运算操作，能够使用8086、8088的所有硬件和软件支持。数值处理的扩充可分为两种类型，一种扩充是使用一个特殊的硬组件——8087数值处理器，而另一种扩充则是以软件为基础的——8087仿真程序（仿真器），显然，前者更为有效，但两者都为8086或8088增加了一些数值数据类型及运算操作性能。8087硬组件与其软件仿真器是完全兼容的。在后面的讨论中，我们将把重点放在8087数值处理器上面。

从原始速率这一观点来看，8087是能够承担复杂的计算任务的第一个微处理器。由于综合了数值分析在过去几年中所取得的成果，从而使得NDP的适用性水平超过了目前的小型机及较大型的计算机中的算术部件。8087的设计目标是相当明确的，即哪怕在程序员不精通数值分析的情况下，直接用手工运算所采用的算法来进行程序设计，它也能得到稳定、正确的答案。这一点看起来好象无关紧要，但有经验的用户却认为这是对机器的基本要求。例如，假设需要把两个单精度的浮点数相乘后再去除第三个单精度的浮点数，即使最后的结果是一个完全有效的32位数，但是多数计算机都会产生溢出错误，而8087却能正确地提供结果。另一个很典型的例子就是在用直接计算法求二次方程的根：

$$\frac{-b\pm\sqrt{b^2-4ac}}{2a}$$

或者在计算含有

$$(1+i)^n$$

的表达式的值时，在大多数机器上都会发生这样一种令人费解的现象，即机器不会始终如一地给出正确的结果。若要求在一切条件下都能获得正确的结果。那么对一般的机器来说，通常就需要使用复杂的数值算法，这对大多数程序员来讲，要求是太高了。然而，一般的应用程序员用直接算法编制出的程序却可以在8087上获得比较可靠的结果。可见，使用8087可以减少为开发可靠、正确的数值计算软件所需要的投资。

8087能够降低软件开发的成本，也能够改进那些不采用实数而以高精度2进制（2^{64}）或10进制（10^{18}）整数进行操作的系统的性能，提高运算精度，减少或去除舍入误差，从而为商业上的应用提供了方便。现实中，精确的算术运算往往是必不可少的，因为看上去不足挂齿的舍入误差也有可能导致金钱上不可估量的损失，甚至会危及人的生命安全。

由于8087的上述特色，因此，可以断言：具备下列任一特点的应用都能够从8087的应用当中获得好处：

1．数值的取值范围很大，或者包含有非整型的数值；

2．算法有可能产生很大或很小的中间结果；

3．计算必须十分准确，即要求有效位数较大；

4．运算能力的要求超过了常规微处理机所具有的能力；

5．要求程序员编出的程序能够提供可靠的结果，而他在数值处理技术方面又不是十分内行。

8086 或 8088 和 8087 结合在一起，对于程序员来讲就好象是一台单独的机器。实际上，8087 为 CPU 增加了一些新的数据类型、增加了寄存器及一些指令。数值处理机能够处理 8 位、16 位、32 位及 64 位的 2 进制整数；18 位的 BCD 整数；还能处理 32 位、64 位及 80 位的浮点实数。在 8087 的内部，所有数据都是以 80 位宽的临时实数格式表示的，这也是 8087 能够提供精确的预期结果的主要决定因素。8087 的取数和存数指令实现操作数类型的转换，即取数指令将操作数转换为临时实数格式，而存数指令则将临时实数转换为所需要的数据类型并存入存贮器单元，这种转换过程对程序员来讲是完全透明的。正象实型操作数能够产生正确的实数结果一样，整型操作数，无论是 2 进制还是 10 进制，也都将产生精确的整数结果，而且当把其它类型的数值转换为临时实数时，不会引起舍入误差。

8087 的内部包括 8 个 80 位宽的数值寄存器，4 个专用寄存器：控制寄存器(16 位)、状态寄存器(16 位)、指令指示器(32 位)和数据指示器(32 位)。8087 中的所有计算都集中在处理器的寄存器栈中进行。8 个 80 位的寄存器的容量等效于 40 个 16 位的寄存器，如此丰富的寄存器空间使得更多的常数和中间结果在计算期间能够保存在寄存器当中，从而减少了存贮器访问的次数，提高了总线的利用率及执行速度。8087 寄存器组的访问方式也相当奇特，该寄存器组可以作为一个堆栈，指令可以隐含地访问其栈顶的一个或两个单元，也可以显式指出所要访问的寄存器，当然这种访问也是相对于栈顶进行的。

8087 具有 63 条指令，按其作用可以分为传送、比较、算术运算、取常数、超越函数计算及控制这六种类型。表 10.1 分类列出了 8087 的主要指令。8087 的汇编语言程序是按 ASM-86 的要求书写的，ASM-86 为所有的指令提供了指令助记符，也为 8087 的各种数据类型提供了相应的伪指令定义。实际上，8087 的指令是与 8086/8088 的指令混合编制在一个完整的程序之中的，即一个程序中的某些指令由 8087 来执行，而另一些指令则由 CPU 来执行，指令的这种分布是机器自动实现的，8087 可以并行地和 CPU 一起对指令进行译码。此外，在 8087 正在执行它的数值指令时，也允许 CPU 继续执行别的指令，利用处理器之间的这种并行性可以提高整个系统的吞吐能力。8086/8088 的所有寻址方式都可用于访问 8087 的存贮器操作数，因而 8087 也能够方便地处理数值数组、结构、基本变量。数值处理机的程序也可以用高级语言 PL/M-86 来编写，PL/M-86 使得程序员在对 8087 芯片不甚了解的情况下，也能编制程序。

表 10.1　8087 的主要指令

分　　类	指　　令
数据传送	取数、存数、交换
算术运算	+、−、*、/、$\sqrt{\ }$、反向减、反向除、换算、求余数、取整、改变符号、求绝对值
比　　较	比较、测试、检验
超越函数	tg、arctg、2^X-1、$Y\cdot\log_2(X+1)$、$Y\cdot\log_2 X$
取常数	0.0、1.0、π、$\log_2 10$、$\log_2 e$、$\log_{10} 2$、$\log_e 2$
处理机控制	初始化、中断控制、存/取控制字、存状态字、状态/环境的保护/恢复、清除事故

8087 还有两个很重要的硬件特点，它们更进一步简化了数值应用的编程。一个特点就是：程序员可以直接用 8087 的指令来引用 8087，而不需要编写指令把 8087 当作某个“I/O 设备”来寻址，也不必为设置 DMA 操作以实现数据传送而付出一定的开销；另一个特点就是：数值

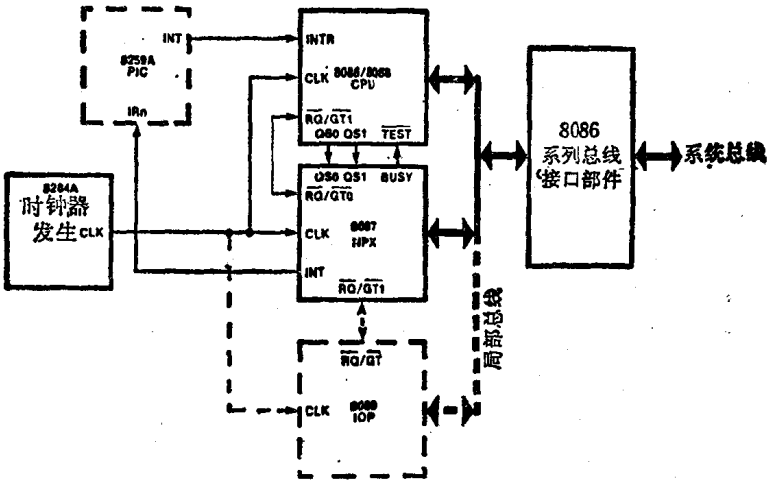


图 10.2 NDP 的组成方式框图

处理机(NDP)能够自动地检测出运行期间出现的异常情况，这些异常情况有可能破坏运算的正常进行，8087 芯片上的异常处理部件可以自动地被调用来处理这些异常事件，并且能够在没有程序的干预的情况下，产生合理的结果，且让运算过程继续执行下去．当然，在检测出某种异常事件时，8087 也能够中断 CPU，从而使其转入用户程序，进行异常事件的处理．

8087 与 8086/8088 之间的配接也是十分方便的，图 10.2 是 8087 与 8086/8088 CPU 连接的一个例子．其中的 8086/8088 需采用最大模式工作．8087 的请求/批准引脚与 CPU 的一个请求/批准引脚相连，从而使得 8087 可以取得对局部总线的控制权，以实现数据传送(取数和存数)．两个芯片的队列状态线(QS_0 和 QS_1)接在一起，这样就使得 8087 能够和 CPU 同步地取得指令并对其进行译码，实际上这也就相当于完成了 CPU 到 8087 的指令传递过程．当 8087 正在执行操作时，它就将 BUSY 置为 1，从图中可以看到，8087 的 BUSY 信号送到了 8086 的 $\overline{TEST}$ 端，所以，CPU 能够利用 WAIT 指令测试这个信号，以便确保 8087 已经准备好执行下一条指令．当 8087 检测出一个异常事件时，它可以中断 CPU，在一般情况下，8087 的中断请求线都是经过 8259A 中断控制器再接到 CPU 上．如图所示，所有处理器(8086/8088、8087、8089)都使用同一个时钟发生器和同一组系统总线接口部件(包括总线控制器、锁存器、收发器以及总线裁决器)，因此，为了给以 8086/8088 为基础的系统增加强有力的数值处理能力，唯一需要加上的硬件就是一片 8087．

在 iAPX 86/88 系列当中，我们把单由 8086 组成的结构命名为 iAPX 86/10，把与 8087 配接后的结构称为 iAPX 86/20，若再加上 8089 就得到 iAPX 86/21，当 8086 与操作系统固件 80130 配接后就形成了所谓的 iAPX 86/30．类似的命名法也适用于 CPU 为 8088 的场合．表 10.2 给出了 iAPX 86 和 iAPX 88 系列的几种结构．

究竟应该选择哪种结构则要根据通用的数据处理、数值运算及 I/O 处理的要求以及系统的性能/价格比来确定．比如说，当要求系统具有较高的数值处理要求时，就应该用 8086 或 8088 与一片 8087 组成 iAPX 86/20 或 88/20 数值处理机；当系统的输入输出要求和数值处理要求较高时，就应将 8086、8087 和 8089 组合起来形成 iAPX 86/21 数值及 I/O 数据处理机，以 8088 作为 CPU 的一个 iAPX 88/21 与 iAPX 86/21 的性能类似，只是由于它使用的是 8 位系统数据总线，因而规模较小，速度也稍低一些，当然价格也较低．在后面的讨论中，86/2X 或 88/2X 代表具有任意个数 8089 的一个数值处理机，也可以用 NDP 来代表数值处理机，NPX

表 10.2　iAPX 86、88 系列中芯片的使用

系 统 名 称	8086	8087	8088	8089	80130
iAPX 86/10	1				
iAPX 86/11	1			1	
iAPX 86/12	1			2	
iAPX 86/20	1	1			
iAPX 86/21	1	1		1	
iAPX 86/22	1	1		2	
iAPX 86/30	1				1
iAPX 88/10			1		
iAPX 88/11			1	1	
iAPX 88/12			1	2	
iAPX 88/20		1	1		
iAPX 88/21		1	1	1	
iAPX 88/22		1	1	2	
iAPX 88/30			1		1

与 8087 代表相同的意思，术语数值指令或数值数据类型表示由 8087 提供的指令或数据类型，术语主机(host)代表 8086 或 8088 微处理器。

§10.2　8087 数值处理器的结构

§10.2.1　8087 的结构概貌

8087 是一个可用于 8086 或 8088 系列的协处理器，它为浮点数据类型、整数数据类型、寄存器及指令系统提供了有力的硬件支持。图 10.3 给出了 NDP 的寄存器集合，图 10.4 则列出了可用于数值指令的七种数据类型。

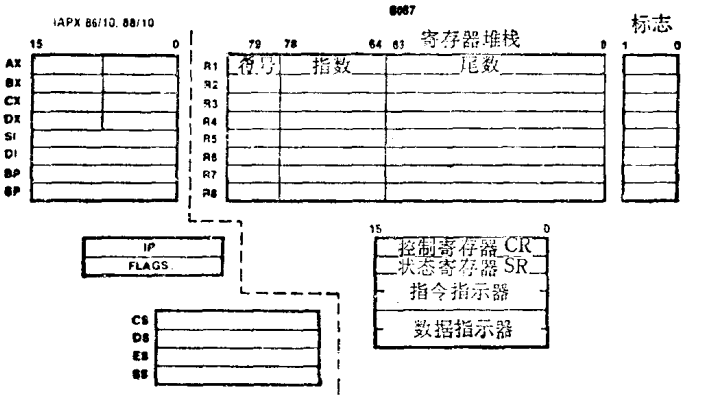


图 10.3　iAPX 86/20、88/20 的寄存器组

每种数据类型都有相应的存取指令。对程序员来讲，8087 芯片和 8087 仿真器具有相同的寄存器及数据类型，程序员利用与使用主机的一般数据类型及指令相同的方法，来使用 8087 的所有数值指令和数据类型。

8087 的数值寄存器使得它在数值运算中能够快速、方便地引用所需要的操作数。在 8087

数据格式	表示范围	精　度	最高有效字节
整数字	10^4	16位	I_{15} … I_0
短整数	10^9	32位	I_{31} … I_0
长整数	10^{19}	64位	I_{63} … I_0
紧凑 BCD	10^{18}	18个10进制位	S — $D_{17}D_{16}$ … D_1 D_0
短实数	$10^{\pm 38}$	24位	S $E_7E_0F_1$ … F_{23}
长实数	$10^{\pm 308}$	53位	S E_{10} E_0F_1 … F_{52}
临时实数	$10^{\pm 4931}$	64位	S E_{14} E_0F_0 … F_{63}

图 10.4　8087 的数据类型

当中，所有的数据都是以临时实型格式(80 位)表示的，从而能够保证运算的精度，也就是说，8087使用的所有数据类型都将被转换成 80 位的寄存器文件格式，存取指令自动地实现这种存贮器操作数的类型与寄存器文件格式之间的转换，取数指令自动地将存贮器中的 16、32、64 位整数，32 或 64 位的浮点数或 18 位的 BCD 数转换成临时实型数，存数指令的转换过程与此相反。存取指令规定了存贮器操作数的格式以及寻址方式。主机中所有的基址寄存器、变址寄存器、段寄存器及寻址方式都可以用来定位 8087 的数值操作数。

在 8087 中，能够访问存贮器数据类型的不仅仅是存取指令，象比较、加、减、乘、除这样一些数值运算指令也都能从存贮器中获得一个 16 位整数、32 位整数、32 位实数或 64 位实数，把它用作指令的一个操作数；但其它的一些数值指令，如平方根、取模、正切、反正切、指数、对数、开方等，却只能使用寄存器操作数。

我们也应该看到，主机与 8087 的寄存器组是分布在两个芯片上的，因而不可能用一条指令在两个寄存器组之间传输数据。若要进行这种类型的传送，则应先将数据送至存贮器单元之中，然后再由目标从存贮器中取出。由于 8087 的 16 位整数的存贮格式与主机是相同的，所以可以较快、较方便地实现这种数据传送操作，但主机不能在一条指令中处理 8087 的其它数据类型，因此，主机程序在读写这些数据类型时必须注意与 8087 所规定的位和字节顺序一致，以免搞错。数值指令也位于主机的指令流中，这些指令的执行也是以它们在指令流中出现的先后为序的。

§10.2.2　8087 的引脚功能介绍

下图给出了 iAPX86/20,88/20 的引脚结构，由于前面已经对 8086/8088 的结构进行了比

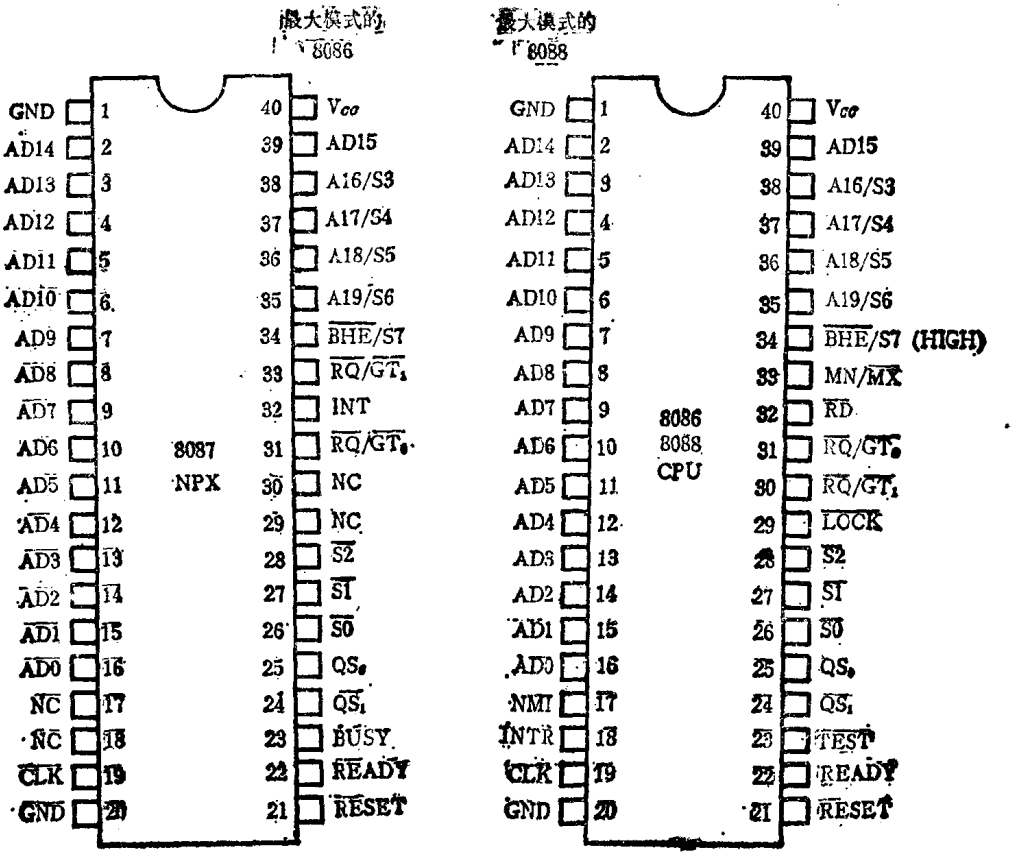


图 10.5 iAPX 86/20、88/20 的引脚图

较详细的介绍，故这里不再重复，下面仅对 8087 的引脚及其功能做些简单的讨论。

AD15—AD0：输入/输出信号

地址/数据总线 AD15—AD0 是多路分时公用的地址(T_1)数据(T_2, T_3, T_W, T_4)总线。A0 对于数据总线的低字节(D7—D0)来说，其作用与 $\overline{BHE}$ 类似，即当要把一个字节数据送到总线的低位部分时，T_1 期间的 A0 就应为低电平，与数据总线的低位部分相连的 8 位设备往往利用 A0 作为片选信号。

A19—A16/S6—S3：输入/输出信号

当进行存贮器操作时，它们在 T_1 期间用作四根存贮器的最高地址线，而在 T_2、T_3、T_W 及 T_4 期间则代表状态信息。若当时的总线周期是由 8087 所控制的，那么 S6、S4、S3 都保持为高电平，而 S5 总是低电平。

$\overline{BHE}$/S7：输入/输出信号

$\overline{BHE}$ 为总线高 8 位可用信号(Bus High Enable)，在总线周期的 T_1 期间，有效的 $\overline{BHE}$ 信号就使得数据能够送到数据总线的高 8 位。与高 8 位数据总线相连的 8 位设备也常把 $\overline{BHE}$ 用作条件片选信号。当要把一个数据字节送往总线的高 8 位时，就应在读写周期的 T_1 期间把 $\overline{BHE}$ 置为低电平。在 T_2、T_3、T_W 和 T_4 期间，可以把 S7 用作状态信息。

$\overline{S_2}$, $\overline{S_1}$, $\overline{S_0}$：输入/输出信号

这是三根状态线，在由 8087 驱动的总线周期内，这些状态线的编码是这样的：

$\overline{S_2}$	$\overline{S_1}$	$\overline{S_0}$	
0	×	×	没用 (unused)
1	0	0	没用
1	0	1	读存贮器
1	1	0	写存贮器
1	1	1	无源(被动状态)

在 T_4 期间,这些状态开始有效,它们在 T_1、T_2 期间也保持有效。当 READY 为高电平时,这些状态线才在 T_3 或 T_W 返回到无源(被动)状态(1,1,1)。8288 总线控制器利用这些状态信息产生所有的存贮器读写控制信号。T_4 期间 $\overline{S_2}$、$\overline{S_1}$、$\overline{S_0}$ 的任何变化都标志着一个总线周期的开始,与之相对应,若在 T_3 或 T_W 期间返回到无源状态则标志着当前总线周期的结束。当 8086/8088 掌握总线控制权时,以上这些信号都由 8087 负责管理。

$\overline{RQ}/\overline{GT_0}$:输入/输出信号

请求/同意信号,8087 利用此引脚从 CPU 获得局部总线控制权,以便传送数据;8087 也可以用该引脚来代表另一个总线主。引脚 $\overline{RQ}/\overline{GT_0}$ 必须与 8086/8088 的两个请求/同意引脚中的一个相连。其请求/同意(批准)的顺序是这样的:

1. 8087 (或连到 8087 $\overline{RQ}/\overline{GT_1}$ 引脚上的某个总线主设备) 发送一个时钟宽度的脉冲给 CPU,表示请求局部总线;

2. 8087 等待同意信号,8087 在接到同意信号之后就开始总线传输(若开始的请求是为另一个总线主设备服务的,8087 就在接到同意信号之后的一个时钟周期内从 $\overline{RQ}/\overline{GT_1}$ 引脚送出该同意信号,而不是立即开始总线传输);

3. 在 8087 的最后一个总线周期完成之后(或从 $\overline{RQ}/\overline{GT_1}$ 上收到一个释放总线脉冲之后),8087 就给 CPU 一个"释放"脉冲。

$\overline{RQ}/\overline{GT_1}$:输入/输出信号

这也是一个请求/同意引脚,它是由某个别的总线主设备用来向 NDP 请求局部总线的。如果在该总线主设备发出使用总线请求时,总线控制权不在 8087 手里,那就在一个周期之后通过 8087 的 $\overline{RQ}/\overline{GT_0}$ 传送请求/同意信号,随后的同意和释放脉冲也通过 8087 传送;如果发出请求信号时,总线控制权在 8087 手中,那么请求/同意序列是这样的:

1. 来自某个其它局部总线主设备的一个时钟周期宽的脉冲表示对 8087 有一个局部总线请求;

2 在 8087 的下一个总线周期的 T_4 或 T_1 期间,8087 发出一个时钟宽的脉冲给发出请求的总线主设备,这就表明 8087 已允许局部总线浮动,且在下一个时钟周期进入"RQ/GT 响应"状态,在RQ/GT 响应过程中,8087 的控制单元在逻辑上是与局部总线分离的;

3. 发出请求的那个总线主设备给 8087 一个时钟宽的脉冲,表示 RQ/GT 请求结束,8087 又可在下一个时钟周期内重新获得局部总线。

由此可见,每次主设备——主设备之间局部总线的切换都要经过由三个脉冲组成的一个序列。

QS1,QS0:输入信号

QS1 和 QS0 给 8087 提供了一个跟踪 CPU 指令队列的状态信息,编码如下:

QS1	QS0	
0	0	无操作，没取出队列中的指令
0	1	从队列中取出操作码的第一个字节
1	0	队列已空
1	1	从队列中取出指令的下一个字节

INT：输出信号

这是一个中断请求引脚，在允许 8087 中断的条件下，此信号表示在数值指令执行期间发生了一个未被屏蔽的异常事件，该信号通常被送到中断控制器 8259A。

BUSY：输出信号

该"忙"信号表示 8087 的数值执行单元正在执行一条数值指令，将它与 CPU 的 $\overline{TEST}$ 引脚连接起来就能够实现 CPU 与 8087 之间的同步操作。若发生了某个没被屏蔽掉的异常事件，那么 BUSY 将保持为高电平，直到清除了该异常事件时为止。

READY：输入信号

READY(准备好)是被寻址的存贮器设备发来的响应信号，它表示该设备已为数据传输做好准备。

RESET：输入信号

RESET(清除)信号使 8087 立即终止它的现行操作。

CLK：输入信号

时钟为 8087 及总线控制器所提供的基准定时信号。

V_{CC}：－5V 电源。

GND：接地

§10.2.3 8087 处理器的结构

前一小节简单介绍了 8087 的引脚及其功能，接下来我们将对 8087 的结构进行讨论。

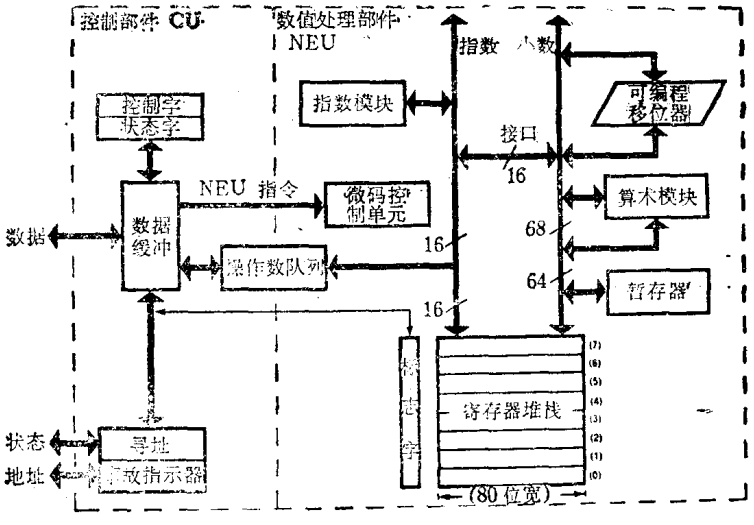


图 10.6 8087 的结构框图

图 10.6 是 8087 的一个结构框图，从图中可以看出，8087 的内部可被分成两个大的处理单元，即控制部件(CU)和数值执行部件(NEU)。其中，数值执行部件 NEU 执行所有的数值指令，而 CU 则起这样一些作用：取指令、对指令译码、读写存贮器操作数、执行 8087 的处理机控制类指令。这两个处理单元能够彼此相对独立地进行操作，在 NEU 忙于处理数值指令时，可使 CU 与 CPU 保持同步。

一、控制部件

控制部件 CU 能够保证 8087 与其主机同步地进行工作，这也是 CU 的主要作用之一。8087 的指令与 CPU 的指令是共存于一个指令流当中的，CPU 从存贮器中取得指令，而 8087 的控制部件可以通过监视 CPU 发出的状态信息($\overline{S_0}$, $\overline{S_1}$, $\overline{S_2}$, S_6)就能够确定 CPU 何时在取指令(状态为 1,0,1,0)，而且，CU 还与 CPU 一道并行地监测数据总线，一旦发现指令字节(或字)在局部总线上变为有效时，它们将一起取得这段指令。

CU 当中也有一个与其主机 CPU 的排队机构相同的指令队列，我们知道，8086 与 8088 的指令队列长度是不相等的，因此，对 8087 的 CU 来讲，它必须能够确定其主机是 8086 还是 8088，进而确定指令队列的长度，这一任务是在总清(RESET)之后，由 CU 通过检查 $\overline{BHE}$/S7 的状态自动地完成的，具体一点就是：当 $\overline{BHE}$/S7 为低电平时表示 CPU 是 8086，否则是 8088。从而，CU 也就可以设置相应的队列长度。通过监视 CPU 的队列状态线(QS_0, QS_1)，CU 又能够和 CPU 同步地从指令队列中取得指令，并及时地对其进行译码。事实上，两个处理器并行地从指令流中取得指令并对指令译码，然后两个处理器分别执行各自的指令。

对 8086 或 8088 而言，8087 的所有数值指令都是以换码指令(ESC)的形式给出的，即这些指令的高 5 位都是一样的，实际上 8087 所能够识别的也仅仅就是这些指令，对于所有的其它指令 CU 将不做任何处理，这是因为这些指令应由 CPU 而不是 8087 来执行。但当 CU 发现一条 ESC 指令时，它就将根据指令的类型来确定是应该自己去执行这条指令，还是把该指令送到 NEU。

数值指令可能要访问存贮器操作数，也可能不需访问存贮器操作数，CPU 必须把那些引用存贮器的指令与不涉及存贮器的指令区别开来。倘若指令要用到一个存贮器操作数，那么 CPU 就将利用指定的某种寻址方式首先计算出该操作数的地址，然后"假读"这个字，其"读"过程与一般的读周期相同，只是 CPU 并不接收所读出的数据，这里的操作数地址可以是 1 兆字节存贮空间的任一单元；如果 ESC 指令不需要访问内存，例如对 8087 寄存器堆栈的操作，CPU 就转去处理下一条指令。总起来讲，根据对存贮器的访问类型，可以把 8087 的指令分为三种类型：(1) 不访问存贮器；(2) 从存贮器中取操作数；(3) 将操作数从 8087 中写入存贮器。对第一种类型的指令，8087 进行的操作比较简单。对于后两种类型的指令，CU 都利用了 CPU 的"假读"周期来取得(并保存) CPU 放到总线上的操作数的地址。如果指令属于第二种类型(需取数据)，那么，当"假读"读出的字操作数在总线上成为有效时，CU 就将取得该数据，若需取出的操作数长于一个字，则 CU 就根据请求/同意规程从 CPU 那里获得总线，并在后继的总线周期内连续地读取操作数的余下部分。对于第三类的 8087 指令(存数据)，CU 与取数过程一样地取得(并保存)操作数的地址，与取数过程不同的是此时 CU 不再接收"假读"出的那个数据字，一旦 8087 做好存数的准备，CU 就从 CPU 那里得到总线，并从已保存起来的地址处开始写入操作数。

二、数值运算执行部件

涉及到寄存器堆栈的所有指令都由数值运算部件 NEU 来执行，这些指令包括算术运算、逻辑比较、超越函数计算、数据传送以及常数指令。NEU 中数据通路的宽度为 80 位，其中小数部分 64 位，指数部分 15 位，另加一个符号位，这就使得内部操作数的传输能以极高的速度进行。NEU 一旦开始执行指令，就把 BUSY 引脚置为高电平，将 BUSY 信号与 CPU 的 WAIT 指令结合起来，就能在 NEU 完成其现行指令之后，重新使两个处理器获得同步。

NEU 的大部分操作都是在寄存器栈中完成的，本节的开头部分曾经对 NDP 的寄存器集合进行过说明，从图 10.3 中也可以看到，8087 的寄存器堆栈中共有 8 个 80 位长的寄存器，其格式是与 8087 的临时实数格式相同的，如下图所示：

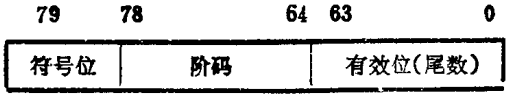


图 10.7　寄存器的格式

在任一时刻，状态字（下面将作介绍）中的 ST 字段总是指向当前的栈顶寄存器。压入 push)操作把 ST 减 1 后再将某个数值装入新的栈顶寄存器；弹出(pop)操作则先从现行栈顶寄存器中取得数值，然后将 ST 加 1。这就使得 8087 的寄存器堆栈与 8086/8088 在内存中的堆栈一样，由高地址向低地址生长。

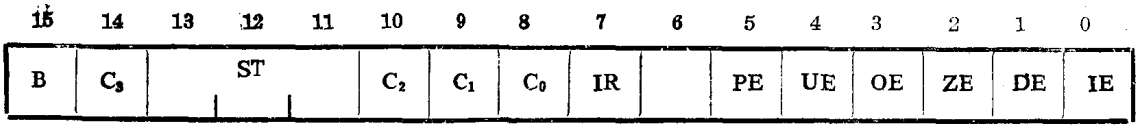
8087 的指令可以隐式或显式地寻址这些寄存器。许多指令都是对栈顶的寄存器进行操作的，这些指令为隐式寻址，它们所访问的是由状态字中的 TP 字段所指向的那个寄存器。例如 ASM-86 中的 FSQRT 指令就是求栈顶数的平方根，并用求得的结果取代原数值，这条指令不需带任何操作数，因为它所要的唯一的操作数就是寄存器栈顶的内容。但也存在这样一些指令，它们允许程序员明确地(显式地)指定所要使用的寄存器，当然，这里所谓的显式寄存器寻址也都是"与栈顶相关的"，即都是相对于栈顶的。在 ASM-86 当中，我们用 ST 代表当前的栈顶寄存器，而让 ST(i)代表栈中从 ST 开始的第 i 个寄存器，其中 $0 \leqslant i \leqslant 7$。例如，假设状态字中 ST 字段的内容是 011B(栈顶为 3 号寄存器)，那么，下面的指令就把 3 号寄存器和 6 号寄存器相加

```
FADD ST, ST(3)
```

寄存器的堆栈结构以及相对于栈顶的寻址方式有效地简化了子程序的编制。其理由就在于，当在寄存器栈上传递参数时，子程序就不必去了解参数究竟放在哪个寄存器当中。同时，也使得不同的例行程序可以调用同一个子程序，而不必遵循在专用寄存器中传递参数的一些惯例(如保护等)，只要栈中还有空，例行程序就可以把参数放入栈中，并调用子程序。子程序以 ST、ST(1)、……的形式来对参数寻址，这就有可能使得子程序在某次调用中作用的是 3 号寄存器，而在另一次调用中却使用了另一个寄存器，例如 5 号寄存器。

接下来我们准备对 8087 的状态字、控制字及标志字进行讨论。状态字反映了 8087 的整体状态。若欲让 CPU 能够检查该状态字，就应先用一条 8087 指令将它放到存贮器当中，然后再由 CPU 用指令来检测状态。实际上，状态字是一个 16 位的寄存器，它被分成几个字段，其格式如图 10.8 所示。

"忙"位(第 15 位)表示 NEU 是在执行一条指令或有一个中断请求(B = 1)呢？还是处于空闲状态(B = 0)；四个数值条件码位(C_0～C_3)类似于一般 CPU 中的标志位，它们的值是由指令



异常(事故)标志位：(为1时表示发生事故)

IE：无效操作

DE：非规格化操作

ZE：除数为0

OE：上溢

UE：下溢

PE：精度

第6位不用(备用位)

IR：中断申请

C_0～C_3：条件码

ST：栈顶指针

B：忙位

ST：000表示栈顶是0号寄存器

001表示栈顶是1号寄存器

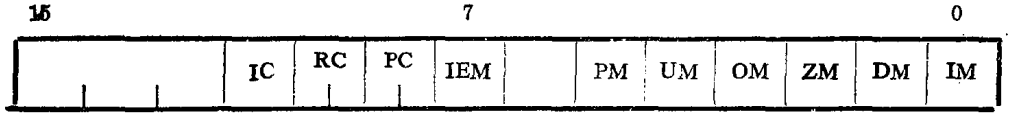
⋮

111表示栈顶是7号寄存器

图10.8　8087状态字的格式

根据NDP操作的结果来设置的；状态字中的ST字段(第11～13位)指向8087的当前栈顶寄存器，一般来讲，刚开始执行时，栈是空的，即ST＝000B，如前所述，每执行一次压入操作都将把ST减1，故此时有ST＝111B，同样，若在ST＝111B时弹出堆栈，则ST加1，从而变为000B，这里需要注意防止产生栈的上溢或下溢错误，状态字的IR字段(第7位)是中断请求标志，只要有一个没被屏蔽的异常事件发生，就把该标志位置为1，即可用它来表示8087向CPU发出了一个待处理的中断请求；状态字的第0～5位用于标志异常事件的出现情况，当在执行某条指令期间，NEU检测出一个异常事件时，就将把这六位中的某一位置为1。

为了满足多种使用的需要，8087提供了几种可供选择的处理方式。用户可以根据自己的实际需要，选择8087按照某种适当的处理方式进行工作。其选择方法也是相当简单的，即从存贮器当中取出一个字，并将其装入所谓的8087控制字当中。当然，该存贮器字必须在装入之前根据控制字的格式以及实际需要预先设置好。图10.9给出了控制字的格式及其各字段所代表的意思：



事故屏蔽位(为1时屏蔽事故)

IM：无效操作

DM：未规格化操作数

ZM：除数为零

OM：上溢

UM：下溢

PM：精度

IEM：中断允许屏蔽：“0”—允许中断；“1”—禁止中断

PC：精度控制：“00”—24位；“01”—保留；“10”—53位；“11”—64位

RC：舍入控制：“00”—最近舍入；“01”—向下舍入；“10”—向上舍入；“11”—截尾舍入

IC：无穷大控制：“0”—投射的(散)；“1”—仿射的(合)

第13～15位：保留位

第6位：保留位

图10.9　8087的控制字格式

由上图可见，控制字也是一个16位的寄存器，其低字节用于8087的中断及异常事件的屏蔽，第0位到第5位是8087所能识别的六种异常事件的屏蔽位；第7位是8087所有中断的屏蔽位；控制字的高字节规定了8087的工作模式，包括精度控制、舍入控制、无穷大控制。利用精度控制位(第8、第9两位)来设置8087的内部操作精度，使8087可以按照低于临时实型精度的方式运行，这在与早期的、较8087精度为低的数值处理机的兼容方面是很有用处的；舍入控制位(第10～11位)规定了运算过程中所采用的舍入方法。

8087为了能够精确地表示运算结果，在其内部还使用了三个附加位(保护位、舍入位和粘性位)，它们对于程序员是完全透明的。如果运算的结果是8087所能够精确地表示的，那么当然不成问题；但如果在算术运算或存取操作中，目标格式不能精确地表示其结果时，那么就会发生舍入的问题。例如，在把某个实数类型的数值送到一个短实数或整数目标中时，就有可能需要对之进行舍入处理。

8087具有四种舍入方式，究竟采用哪种舍入方式则由控制字中的RC字段来确定。假设结果b不能用目标数据类型精确地表示，而在给定的数据类型中与b最为接近的两个可表示的数为a和c，且设a<b<c，那么，8087在得到结果b之后，就将根据舍入控制字段所选定的方式把b舍入成a或c，舍入结果如表10.3所示。

表10.3　舍入方式

RC字段	舍入方式	舍入动作及结果
00	最近舍入	从a,c中取较接近于b者为值，若一样接近，取最低有效位为0者作为结果
01	向下舍入(趋向$-\infty$)	舍入结果为a
10	向上舍入(趋向$+\infty$)	舍入结果为c
11	截尾(趋向0)	a,c中绝对值较小者为舍入后的结果

在实际应用中，舍入方式的选择是以应用的实际需要为基础的。在大多数应用当中，通常都是采用“最近舍入”方式，它与我们所习惯的“四舍五入”方法甚为相似，确实，这种舍入方式为结果真值提供了最准确且在统计上没有偏差的一个估计值；“截尾”舍入方式往往用于对整数的运算当中；“向下舍入”和“向上舍入”方式都是直接的舍入方式，可以把它们用于区间运算。

此外，8087的控制字中还有一位(第12位)专门用来对无穷大数值进行控制。8087的实数系统有两种模型，一种是仿射闭包，还有一种叫投射闭包，其图解方法表示在图10.10中。

控制字中的IC字段用来从这两种模型中选出一个来，当IC字段设置为0时，则表示选中投射闭包，此时8087的特殊值无穷大没有正负之分，即它是不带符号的，在绝大多数计算机当中，使用的都是这种数值系统模型；当IC字段的内容为1时，则表示实数系统为一个仿射闭包，此时的无穷大有$+\infty$和$-\infty$之分，显然它要比投射闭包提供了更多的信息，但在一些实

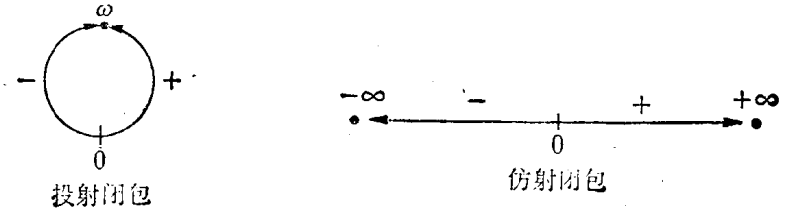


图10.10　无穷大模型

际应用当中，符号反而有可能引起一些不必要的麻烦。例如，某个算法以不同的执行方法得到的中间结果X的值分别为 +0 和 −0，这在一般的系统当中都认为是正确的，但如果后面要在仿射的实数系统中计算 1/X 的值，那么就会出现这种现象：对数值相等的X进行相同的运算得到的却是两个完全不同的值（+∞ 和 −∞）。然而，我们也应该看到，尽管投射闭包提供的信息比较少，可是它却不会带来这些不利影响。因此，在通常情况下，全局上总是使用投射方式，而把仿射方式作为局部计算的一种备用方式，以便程序员在这些局部计算中，利用符号的优点，清楚地了解运算的执行情况，且不会导致错误的结果。

在状态字和控制字当中，都包含有六个对应于8087的异常事件的信息位。8087在其指令执行期间，都将检测这六种异常条件。如果这些异常事件没被屏蔽掉且允许中断出现的话，那么，任何一种异常事件的发生都将引起一次中断请求；若不允许这些中断，那么8087就将继续执行下去，而不管主机是否已清除了该异常事件；倘若发生的是某个已被屏蔽的异常事件，那么8087就在其状态字中做上相应的标志，并且完成一个片上的异常处理过程，同时继续执行下去。下面讨论一下8087的六种异常情况。

1．无效操作（Invalid operation）：在出现下面某种情况时，8087就将报告一次无效操作：堆栈上溢、堆栈下溢、将某个非数值数据(NAN)用作操作数、结果为不定型（0/0，∞−∞，求负数的平方根等等）。无效操作一般说明有一个程序错误。

2．上溢(Overflow)：如果对于目标操作数来讲，结果的数值太大，以至于目标的规定数据格式容纳不下，那么8087就将发出上溢信号。

3．下溢(Underflow)：与上一种情况恰恰相反，尽管结果不等于0，但其数值太小，规定的数据格式不能表示它，在这种情况下，8087将发出下溢信号。

上溢和下溢均表示操作结果超出了目标数据格式的表示范围，在多数算法当中，这类"太大"或"太小"的数值更容易在计算中间结果时产生，由于在8087中，中间结果常常是以临时实数格式表示的，它的取值范围比较大，所以相对讲，这种上溢或下溢情况在有8087的系统中是比较少见的。

4．分母为0（Zero divisor）：当用0去除一个非0的操作数时，8087就指出除0事故。

5．非规格化的操作数(Denormalized operand)：当操作数或结果当中至少有一个为非规格化数据时，就说是发生了该异常事件。

6．结果不精确（Inexact result）：如果操作结果的真值不能精确地以规定的目标格式表示出来，那么，8087将根据规定的舍入方式对该结果进行舍入处理，同时设置精度事故标志。在实际应用中，这类事件是经常发生的，它表示在运算过程中牺牲了一些精度，这通常是可以被接收的。

在以上这六种异常事件当中，无效操作、被0除、非规格化操作数这几种事故是在执行实际操作之前被查出的，而上溢/下溢及精度事故等则要等到真实结果被计算出来了之后才能发现，因此在发现前面一类事故时，寄存器堆栈和存贮器中的内容不会改变，它们与引起错误的指令未被执行时的状态一样；然而，在发现后一类事故时，寄存器堆栈和存贮器的内容可能已被更新，指令的执行已告完成。由此可见，对这些异常事件所需进行的恢复与处理也是不相同的。

在多个异常事件同时发生的情况下，对异常事件的处理是按下列的优先顺序依次进行的：

非规格化操作数(未屏蔽)

无效操作

被 0 除

非规格化操作数(被屏蔽)

上溢/下溢

精度

在发生某个异常事件时,8087 就把状态字中的一个相应位置为 1,以指示该异常事件。紧接着,8087 就检查控制字中与之相对应的事故屏蔽位,以便确定它是否应该发出一次中断请求,从而调用事故处理程序。当屏蔽位等于 1 时,表示该事故已被用户软件所屏蔽,此时,就由 8087 本身在其芯片上执行对该事故的屏蔽响应(缺省的异常事件处理过程),8087 总是在屏蔽响应产生了一个标准结果之后,才继续执行下面的指令;而当屏蔽位等于 0 时,表示该事故未被屏蔽,此时,8087 就要执行未屏蔽的响应程序,未屏蔽的响应总是通过中断 CPU(假设中断通路畅通),调用用户软件提供的故障处理程序,对发生的事故进行处理。这些响应汇总在表 10.4 当中。

表 10.4 异常事件的响应

异常事件	屏蔽响应	非屏蔽响应
无效操作	可把 NAN 当作结果	申请中断
被零除	返回一个代表无穷大的码字,其符号是两个操作数的异或值	申请中断
非规格化	处理过程照常进行(有可能需要调整)	申请中断
上溢	结果为代表无穷大的码子	寄存器目标:调整指数,使其落入可表示的范围内,存贮结果,申请中断. 存贮器目标:申请中断
下溢	将小数部分右移,直到指数部分落入能表示的范围内(非规格化)	同上
精度	返回经舍入处理的结果	送回舍入结果,申请中断

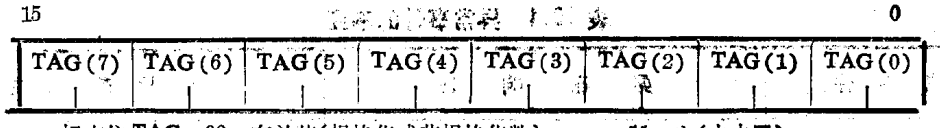
8087 状态字中的事故标志只能由 FCLEX 指令清除,FRSTOR 或 FLDENV 指令也可以修改这些标志。程序可以检查状态字,以检查在计算过程中是否发生过事故。顺便提一下,8087 另外还有一组内部事故标志,它们在每条指令执行之前都将被清除,实际上,促使 8087 去进行事故响应的正是这组标志,而不是状态字中的标志。状态字中的标志位仅仅为程序员提供了一个事故的累计记录。

通过把控制字中的事故屏蔽位置为不同的数值,就能够确定处理事故的责任是应由用户承担,还是应由 8087 自己完成。一般来讲,事故处理程序往往是难于编写的,"好"的事故处理程序更是如此,值得庆幸的是,8087 的屏蔽响应是为针对各种异常条件提供"最合理的"结果而专门设计的,从大量的应用当中我们发现:在把除了无效操作之外的所有事故都屏蔽掉之后,就能够以最少的软件投资获得令人满意的结果。在此,之所以没有屏蔽掉无效操作这一异常事件,是因为它通常代表了一种程序中必须加以纠正的"致命"的错误。倘若 8087 对某个事故执行了一次非屏蔽响应,那它首先就得向 CPU 发出中断请求,然后执行一段用户编写的事故处理子程序。也就是说,8087 先把状态字中的中断申请位(IR)置位,但这并不能保证能够

立即中断 CPU，因为中断请求可能会被 8087 控制字中的中断允许/屏蔽位(IEM)、8259 中断控制器或 CPU 所堵塞。因此，如果发生了某个没被屏蔽的异常事件，则最关键的是打通通向 CPU 的中断请求通路，以便用户软件能够找到事故发生时的现场，并执行相应的处理程序。这些由用户所编写的事故处理程序是以 8086/8088 中断处理过程的形式出现的。虽然事故处理程序之间差别很大，但是，它们一般都具有以下一些基本步骤：

1. 保护事故发生时 8087 的现场；
2. 清除状态字中的相应事故位；
3. 让 CPU 可以接受中断；
4. 通过分析原来的状态字及控制字，识别事故的类型；
5. 对该具体的事故进行处理(各事故处理程序之间的主要差别就在这里)；
6. 返回断点，继续执行原来的程序。

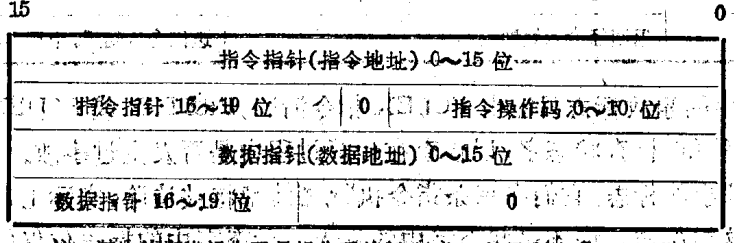
8087 还有一个标志字，它是 8 个数值寄存器的内容的标志，其格式如图 10.11 所示。标志字的主要作用就是优化 NDP 的性能，程序员不必关心它。



标志位 TAG：00—有效值(规格化或非规格化数)　11—空(未占用)
10———特殊值　01—0 值

图 10.11　8087 的标志字格式

为了有利于用户编写事故处理程序，8087 还提供了一个指令指针和一个数据指针，有时我们也把它们统称为事故指示器，其格式如图 10.12 所示。每当 8087 执行一条 NEU 指令时，CU 就把指令地址、操作数地址(若该指令含有存贮器操作数的话)以及指令的操作码保存到事故指示器当中，由于 8087 的指令能够把这些数据存入内存，所以，事故处理程序也就可以获取这些信息，从而了解有关引起事故的那条指令方面的情况。



注：其中的指令操作码是操作码的低 11 位，其最高的 5 位是固定的(11011B)，故无需保护

图 10.12　8087 的指令和数据指针

§10.2.4　8087 的数字系统

从理论上讲，人用笔和纸所进行的计算可以是完全精确的，而不带任何误差。也就是讲，人所使用的实数系统是连续的，任意大小及精度的数值都可以被表示出来，在这样的一个实数系统中，对数值的大小以及数值的精度(即数的有效位数)不存在上限或下限。对任何实数值来讲，总存在着无穷个较大的数值和无穷个较小的数值；同样，在任何两个实数之间，也存在着

无穷个实数，例如，在 2.5 与 2.6 之间，就存在着 2.5001023、2.53、2.5879……这样的无穷个实数。当然，人们也很希望计算机能工作于整个实数上，但这在实际上是不可能的。不管计算机有多大，其寄存器及存贮器的长度总是有限的，这就使得计算机所能表示的数值大小(范围)及数值精度受到限制。实际上，计算机的实数系统是一组离散的、有限的数值，它仅仅是实数的一个子集，是实数系统的一种近似。因此，在设计计算机的数字系统时，重要的是考虑如何设计一个满足具体要求的近似的实数系统，而不是企图设计无限的、连续的实数系统。

8087 实数的表示范围大约为 $\pm 4.19\times10^{-307}\sim\pm1.67\times10^{308}$，在实际应用当中，需处理的数据和最终结果超出这一范围的情况是相当罕见的，也就是说，8087 已为实际应用提供了一个"足够大"的取值范围。表 10.5 列出了 8087 实型数的取值范围，从表中可以看到，尽管 8087 是微机当中的一个协处理器，然而它所提供的数值取值范围是相当大的。

表 10.5　8087 中实数的取值范围

数　据　类　型	近似的取值范围
短实数(单精度)	$8.43*10^{-37}\leqslant\|X\|\leqslant3.37*10^{38}$
长实数(双精度)	$4.19*10^{-307}\leqslant\|X\|\leqslant1.67*10^{308}$
临　时　实　数	$3.19*10^{-4932}\leqslant\|X\|\leqslant1.2*10^{4932}$

图 10.13 把 8087 的基本实数系统投影到实数轴上，其中的实点(·)表示 8087 所能精确表示的实数。

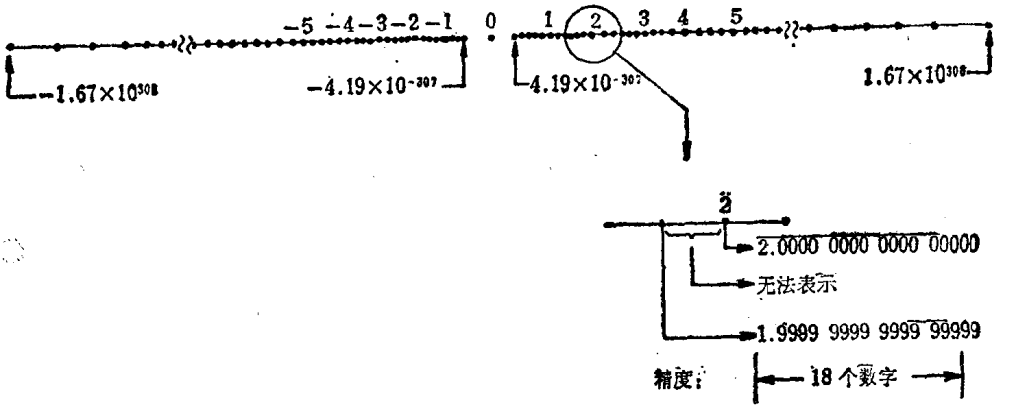


图 10.13　8087 的实数系统

8087 的两个相邻的实数之间总有一个间隔，这也在图 10.13 中得到了说明。如果某次运算的结果正好是 8087 所能表示的某个实数值，那么 8087 就精确地表示它；但常会出现这种情况，结果值落在某两个相邻的实数之间，此时，8087 就要根据舍入规则(见 §10.2.3)将该结果舍入成为它所能够表示的数。因此，若要表示一个位数比 8087 所能容纳的位数多的实数(例如，一个 20 位的 10 进制数)，就得牺牲一点精度。此外，从图 10.13 还可以看到，8087 所能够表示的实数不是均匀地分布在实数轴上的，在任意的 2 的连续的幂次方之间，8087 所能表示的实数的个数是相等的，即在 65536 与 131072($2^{16}\sim2^{17}$)之间存在着多少个可表示的实数，那么在 2 和 4($2^1\sim2^2$)之间也就存在着多少个可表示的实数。因此，可表示的两个相邻实数之间的间隔是随着数值的增大而增大的。

8087 在其内部操作中所采用的是临时实数格式，临时实数又将 8087 的取值范围大约扩展到 $\pm3.4\times10^{-4932}\sim\pm1.2\times10^{4932}$，临时实数的精度约为 19 位的 10 进制数。提供临时实数

类型的目的是想给常数及中间结果以更大的表示范围和更高的精度，而不是用于表示数据或最终结果。从实用的观点来看，8087 的实数集已经是足够大、足够密的了，即使是与包括大型计算机在内的系统相比，8087 实数系统的近似程度也是相当高的。当然，这并不是说 8087 能够精确地表示所有的实数，它还远不能说是一个精确的实数表示系统，不过计算机中的实数运算本来就是近似的。与此不同，8087 在其整数子集上的算术运算却可以是精确的，也就是说，只要两个整数的运算结果是一个整数，并且其结果落在取值范围之内，那么，8087 就一定能够精确地把该结果表示出来。例如：4÷2 产生的是能精确表示的整数结果，而 1÷3 则不然。

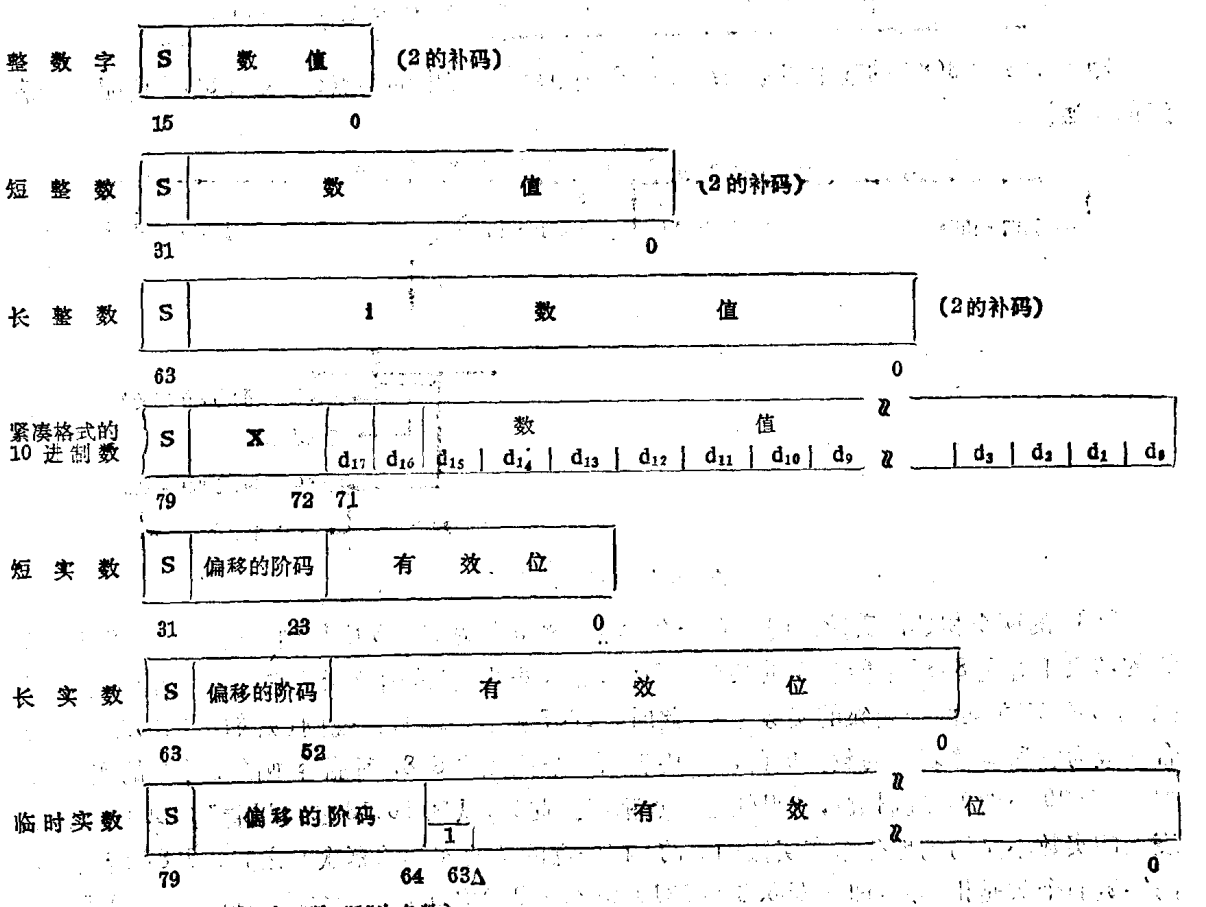
在 §10.2.3 中，我们曾经给出过 8087 所提供的数据类型，实际上，也可以把 8087 所能识别的数据类型分为三类：

2 进制整数；

紧凑格式的 10 进制整数；

2 进制实数。

图 10.14 扼要地表示出了各种数据类型的格式，其中，最左边的位为最高有效位。表 10.6 给出了各种数据类型的取值范围及有效位的位数(以 10 进制表示)。



注：S=符号位("0"为正数，"1"为负数)

d_n=10 进制数(2位/字节)

X=无效位

△=隐含的 2 进制小数点的位置。对临时实数是被存贮的，而在短的和长的实数中是隐含的。指数的偏移：短实数：127(7FH)，长实数：1023(3FFH)，临时实数：16383(3FFFH)

图 10.14 8087 的数据格式

表 10.6　8087 的几种数据类型

数据类型	位　数	有效数位(10进制)	近似范围(10进制)
整数字	16	4	$-32768 \leqslant X \leqslant 32767$
短整数	32	9	$-2*10^{9} \leqslant X \leqslant 2*10^{9}$
长整数	64	18	$-9*10^{19} \leqslant X \leqslant 9*10^{19}$
紧凑的10进制数	80	18	$-\underbrace{99\cdots99}_{18} \leqslant X \leqslant \underbrace{99\cdots99}_{18}$
短实数	32	6～7	$8.43*10^{-37} \leqslant \lvert X\rvert \leqslant 3.37*10^{38}$
长实数	64	15～16	$4.19*10^{-307} \leqslant \lvert X\rvert \leqslant 1.67*10^{308}$
临时实数	80	19	$3.19*10^{-4932} \leqslant \lvert X\rvert \leqslant 1.2*10^{4932}$

1．2 进制整数

8087 的 2 进制整数有三种不同的格式，实际上，它们除了长度不同之外，其表示格式都是一样的。最左边的一位是符号位，它也与通常的规定相同，符号位为 0 表示它为正数，为 1 表示负数，负数是以 2 的补码格式表示的，值得指出的是，8087 中只有负的 2 进制整数是用补码表示的。数值 0 被表示为正数（所有位皆为 0）。8087 的整数字格式是与 8086/8088 的 16 位带符号整型数格式相同的，这为两者之间的数据传送提供了方便。由于这三种 2 进制整数的长度不同，故它们的取值范围也是不一样的。

2．10 进制整数

在 8087 当中，10 进制整数是以紧凑格式存放的，具体讲就是：除了带有符号位（"0"为正、"1"为负）的最左边的那个字节之外，其余的每个字节都代表了两个 10 进制数(BCD 数)，每四个 2 进位代表一个 10 进制数，其值在 0～9 当中。这里的负数不再以补码格式存放，负数与正数的区别仅仅在于符号位的不同。

3．2 进制实数

8087 的 2 进制实数格式由三个部分组成：符号字段、阶码字段和有效位字段。这种表示格式类似于通常的科学计算表达形式。符号字段的作用仍是指出数的正负，有效位字段用于存放数值的有效数字(尾数)，阶码字段用于调整二进制小数点的位置，它决定了数值的大小。与十进制数类似，实数的正负之间的区别也只在于它们的符号字段的内容不同（"0"为正，"1"为负）。

8087 通常是以规格化的格式来表示其有效数字的，即以 1△fff……ff 的格式表示有效数字。其中的△表示一个假设的小数点，故有效数字由一位整数及一个由多位组成的小数部分组成，小数部分所包含的位数是随着实数类型的不同而不同的，对于短实数来讲此位数为 23，对长实数是 52 位，对于临时实数则为 63 位。在实数的这种规格化表示中，整数位取值总是"1"，这样就消除了"小"的数值前面的那些"0"，从而使得有效位字段所表示的有效位的数目达到最大值。有趣的是，8087 的短实数和长实数格式中的整数位是一个隐含位，即它实际上并没有真正出现在实数格式当中，而只有在临时实数格式当中，整数位才真正存在。

倘若实数只能以上述形式出现，那么所有经过规格化处理的实数的值都在 1 到 2 之间，这显然是不能满足实际应用的需要的，如果说规格化处理是为了提高数值精度的话，那么引入指数字段的目的则是为了扩大数值的表示范围。指数字段把 2 进制小数点定位到有效数字当中，它与科学计算中所采用的 10 进制指数类似，指数为正表示小数点应往右移；指数为负则使 2

进制小数点左移，此时，根据需要要在前面加上一些0；指数为真0时，表示原假定的2进制小数点的位置也就是实际的小数点的位置。由此可见，指数字段决定了实数值的大小。为了简化实数之间的比较，8087则以偏移的形式来存放指数，即在原真指数上加上了一个常数值——偏移基数。这个偏移值对于不同的实数格式是不相同的，由图10.14可见，对应于短实数的指数偏移值为3FH，长实数的指数偏移值是3FFH，临时实数的指数偏移值则为3FFFH。从这些对偏移值的选择中可以看出，选择这样一些偏移值的目的是想使各自的指数部分都为正，这样做的好处就是：两个实数可以象两个不带符号的2进制整数那样来进行比较。当它们从指数的最左位开始逐位向右进行比较时，一旦发现某个对应位不同时，数的大小顺序就确定了，也就没有必要继续对后面的各位进行比较。采用了偏移指数之后，实数的真实指数可以从对应的指数字段中减去相应的偏移基数来求得。

长、短实数格式只存在于内存当中，当把它们装入寄存器时，8087就自动地将其转换为临时实数，即转换为内部操作数格式。同样，在把寄存器中的数据送入内存时，8087又自动地将临时实数转换为目标数据类型的格式，这里的目标数据类型可以是短实数、长实数，也可以是临时实数，存贮器中的临时实数通常都是中间结果。大多数应用都使用长实数格式来保存实数数据的结果，长实数格式为结果的表示提供了足够的精度及取值范围，由此得出的结果在绝大多数情况下都是令人满意的，程序员通常也乐于采用这种表示方法。在存贮器容量感到比较紧张的系统当中，往往采用短实数格式，当然这会在某种程度上降低数值的精度，减小数值的取值范围，从而带来一些安全性方面的问题。临时实数格式一般用于存放中间结果，它能够有利于避免在计算中间结果时由于舍入或上/下溢所引起的麻烦。如果把临时实数格式用于保存数据或提供最终结果，那么也就失去了8087内部所构成的安全特点，更何况对于大多数微型机的应用来讲，由长实数格式所提供的取值范围和数值精度已是足够的了，因此临时实数格式一般只用来表示中间结果。

综合上面的讨论可以看出，8087的各种数据类型都具有一定的典型用途，总结如下：

16位整数——变址、循环计数及小型程序的控制值；

32位的短整数——一般的整数运算；

64位的长整数——扩大范围内的整数运算；

18位的2—10进制数——商业应用及10进制算术运算；

32位短实数——为了节省存贮器而减小取值范围和降低精度的实数运算；

64位的长实数——提倡使用的浮点数变量的类型；

80位的临时实数——中间结果的表示格式或用于某些高精度运算当中。

表10.7 实数的记数方法

记数法	数值		
原10进制数	178.125		
科学计算10进制数	1Δ78125E2		
科学计算2进制数	1Δ0110010001E111		
科学计算2进制数(偏阶)	1Δ0110010001E10000110		
8087的短实数(规格化的)	符号	偏移阶码	有效位
	0	10000110	1Δ01100100010000000000000(隐含的)

作为例子，表10.7列出了一系列用来表达同一数值(178.125)的记数方法，借此说明各种数据格式之间的转换方法。当然，并非所有的10进制数都可以用2进制精确地表示出来，例如，10进制数0.1就不能精确地用2进制来表达，当ASM-86或PL/M-86的翻译程序遇到这类数值时，就将产生一个经舍入处理过的2进制近似值。

8087的各种类型的数据都可以存放到存贮器当中，它们在存贮器中的存放格式如图10.15所示。符号位总是位于地址最高的存贮器字节中（最高位为符号位），最低有效位所处的单元地址也最低。16位整数字与8086/8088的16位带符号整数的存放格式是完全一样的，且可直接由CPU或8087的指令所引用。

8087能对其CPU的一兆字节的存贮空间中的任何单元进行访问，存贮器操作数的地址是由CPU形成的，8087可以使用CPU的各种寻址方式来访问存贮器操作数。8087还有几条特殊的存贮器访问指令，用于访问处理器的控制字和状态字。主机总是把8087的指令当作ESC指令，若8087的指令要求访问存贮器操作数，那么主机就利用它的某种寻址方式——直接寻址、寄存器间接寻址、基址寻址、变址寻址或基址变址寻址——形成存贮器操作数的有效地址，该地址是对应的操作数的最低单元地址，即它指向操作数的最低字节。由于8087的操作数能以8086/8088的任一种寻址方式来访问，所以它也与主机类似，具有容易使用象数组、结构、表格等数据结构的特点。

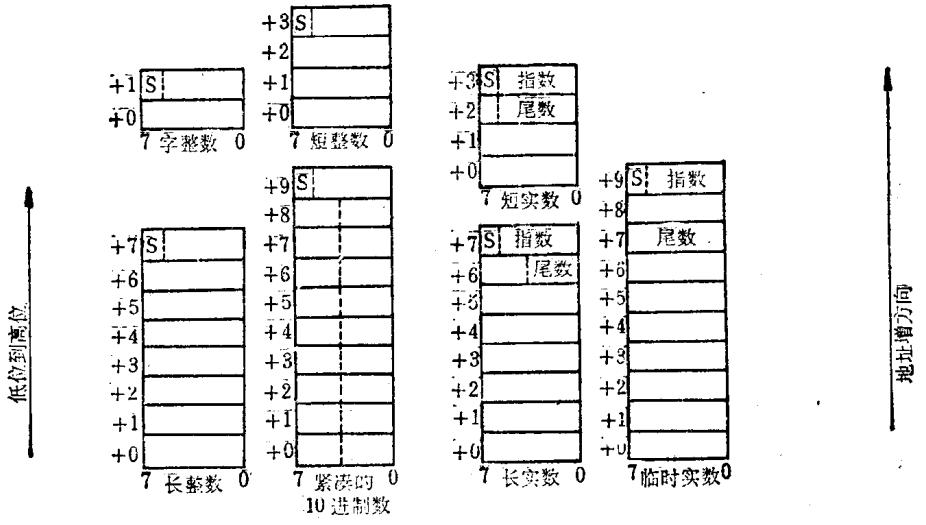


图 10.15 各种数据在存贮器中的存放格式

主机在形成了存贮器操作数的实际地址之后，就发出这样一个20位的地址，接着8087就捕获这个地址并把它保存起来。如果指令是要求从存贮器中取入数据，那么8087就将在主机的"假读"过程中，一旦发现数据（最低地址的字）在总线上成为有效时，就取得该数据。根据主机的类型(8086或8088)和操作数在存贮器中的位置（偶地址或奇地址），主机的"假读"过程可能需要一个或两个总线周期，在"假读"过程完成时，8087仅仅取进其操作数的最低字，由于8087的操作数（整数字除外）都是由若干个字组成的，为了能够读进整个操作数，8087就按照请求/同意($\overline{RQ}/\overline{GT}$)规程发出使用局部总线的请求，在它从CPU那里获得局部总线之后，它就不断地递增保存起来的存贮器操作数地址，在相继的总线周期内读进操作数的其余部分，然后把局部总线归还给CPU，至此，整个存贮器操作数已全部取进，8087也就开始执行其指令。倘若指令是要求把操作数从8087送存到存贮器，那么8087就与CPU一样，也不理睬由CPU

"假读"出的数据，这就意味着 CPU "假读"的作用仅仅是把操作数的地址通知 8087。此时，一旦 8087 做好了把指令的结果写入存贮器的准备，它就向 CPU 发出局部总线的使用请求，利用请求/同意规程从 CPU 得到总线，并在后继的总线周期内连续地写完操作数，然后把总线控制权还给 CPU，完成整个写动作。

在 §10.2.3 中我们曾经介绍过，8087 能够自动地确定其主机是 8086 还是 8088，那时解决的主要是指令队列的长度设置问题，大家知道，8086 与 8088 最主要的区别是它们的数据总线宽度不一样，8086 为 16 位，8088 则为 8 位，因此，当 8087 与 8088 相连时，8087 就将按照其主机 8088 的数据传送方式，每个总线周期传送一个字节。而当 8087 与 8086 一起使用时，它又象 8086 那样进行数据传送：以两个总线周期访问位于奇地址的字，以一个总线周期访问位于偶地址的字，当 8087 从 8086 的存贮器中读/写多个字的奇地址操作数时，它就在第一个总线周期内读写一个字节，从而使地址变为偶数，然后以字为单位读写操作数的"中间部分"，最后读/写操作数的最末一个字节(最高字节)，这样，从某种意义上讲，就优化了数据的传送过程。由上面的讨论可以看出，为了减少操作数的传送时间，使 8087 少用一些系统总线周期，最好把 8087 的存贮器操作数定位在偶数地址上（利用 ASM-86 的伪指令 EVEN 就能实现该功能），即使 CPU 是 8088，我们也提倡这样来定位操作数，因为这样一来，当把 8088 换成 8086 时，就可以获得最高的数据传送效率。请注意这样一个事实：8087 的所有数据类型都是由偶数个字节组成的。因此，把其操作数定位在偶地址上就更显得有利可图了。此外，还有一点在寻址操作数时是应该注意的，那就是尽管 8087 本身不需要把任何存贮器单元划为其专用区域，但它必须注意那些与 CPU 及 8089 有关的存贮器专用保留区，因为使用这些保留区中的任何单元都有破坏当前或今后 Intel 公司软/硬件产品的兼容性的可能。

8087 除了能够表示上述各种类型的正数、负数之外，还可以表示几个"特殊的"实体，这些特殊值的引入使得 8087 更加灵活，其适用范围更大，对于大多数用户来讲，不需要了解这些细节，因而这里只对它们简单地讨论一下。数值 0 在实数和 10 进制整数格式当中有正、负之分，而在 2 进制整数格式当中，0 的符号总是正的；利用实数格式可以表示特殊值 $+\infty$ 和 $-\infty$，当发生用 0 除、结果超过目标的取值范围这类事件时，8087 就将产生无穷大作为结果，以此作为它对异常事件的内部响应，在 8087 当中，特殊值无穷大可以参与算术运算和比较操作；倘若程序员企图进行一次不能给出合理结果的操作，那么 8087 或者发出一个中断请求，让程序员对此事件进行处理，或者送回一个特殊值——无定义——作为结果，求负数的平方根就是一种无效操作，每种数据格式都有一个代表特殊值"无定义"的编码。在这种事件发生时，一个可取的解决方法就是停止当前的计算过程，发出中断请求，转向用户为此编写的事故处理程序。当然，也可以采用另一种解决方式，即让程序继续执行下去，但此时，特殊值"无定义"就将在计算的过程中传递下去，当把它作为运算的最终结果给出时，就将指出计算过程发生错误。

在 8087 的实数格式当中，有一个专用于表示特殊值 NAN(非数值)的值域，它们代表各种正的或负的 NAN，上面介绍的特殊值"无定义"就是其中的一个码子。很显然，当把某个非数值数据 (NAN) 用作 8087 指令的一个操作数时，将发生一次无效操作事故。NAN 的一个很重要的应用就是检查未经定义过的变量。当这类无效操作事件发生时，可以利用陷阱暂停 8087 的执行，转向用户程序，对 NAN 进行处理；也可以由 8087 的内部事故处理程序对此进行处理，它将把 NAN 本身作为操作的结果，即这里的 NAN 在计算过程中是传递的，所得到的最终结果也只能是 NAN。

前面讲过，8087 的实数是以规格化的浮点格式表示的，此外，8087 还可以保存那些非规格化的实数，并能对这类实数进行操作。所谓非规格化实数，是指那些在有效数字前面有一个或多个 0 存在的实数。当计算所得结果是一个极小的数值，小到无法用规格化形式表示时，就只得用非规格化形式表示之。虽然由于前导 0 的存在，数值的有效位数相对减少，从而损失了一些精度，但它为"小"数值的表示提供了可能性。这里值得注意的是，在常规的算法当中，特别小的数值一般是作为中间结果出现的，很少作为最终结果，而在 8087 中，中间结果是以临时实数格式表示的，它能够表示小至 $\pm 3.4\times 10^{-4932}$ 这样的数值，可见，在实际应用当中，8087 中的非规格化数是极少会出现的，然而为了保险起见，8087 还是提供了以非规格化实数的形式进行存数、取数及操作的手段。

§10.3 8087 的指令系统

按照指令的功能，我们把 8087 的指令系统分为六个大类，即：

数据传送指令

比较指令

算术运算指令

超越函数计算指令

常数指令

处理器控制指令

典型的 8087 指令接收一个或两个操作数作为输入，对输入信息进行适当的处理，产生一个结果作为输出。操作数一般是寄存器或存贮器单元中的内容，有些指令的操作数是特定的，例如，FSQRT 的操作数总是寄存器栈顶的内容；而另外一些指令则允许或要求程序员明确地给出所需要的操作数；还有一些指令能够接收一个显式操作数和一个隐式操作数（通常是寄存器堆栈顶的内容）。不管操作数的来路为何，它们也都可以被分成源操作数和目标操作数两种基本类型。源操作数仅仅是向指令提供的一个输入信息，它不被指令修改，即使某条指令需要把源操作数从一种格式转换为另一种格式时（如：实数变为整数），也不会修改源操作数，因为这种转换过程是在内部工作区内进行的。目标操作数虽然也可以向指令提供一个输入信息，但它和源操作数是有区别的，因为它被用来接收操作结果，故其内容往往被指令改变。

本节将逐条说明 8087 的所有指令，对于允许使用多种形式操作数的指令不再分开介绍。例如：对于指令 FADD 来讲，它有三种操作数组合形式，即不写操作数、只写出一个操作数、写出一个目标操作数及一个源操作数。在碰到这种情况时，我们用斜线"/"来代替"或者"的意思，即用它作为可选择的操作数形式之间的分界符；而用双斜线"//"表示可以选择隐含的操作数形式，当然，在程序当中，不应写出这种斜线或双斜线。根据这些规定，我们在介绍 FADD 指令的时候，将把它说明为：

FADD//源/目标，源

这就意味着 FADD 指令可以以下面三种形式出现

FADD

FADD 源

FADD 目标，源

§10.3.1 数据传送指令

数据传送指令实现寄存器堆栈之间、寄存器堆栈与存贮器之间的操作数传送功能，取数指令可以将存贮器中的整型数、实型数、BCD (2—10 进制)数和临时实型数取到寄存器栈顶，且将各种类型的操作数转变为临时实数格式；存数指令的作用与此相反。存取指令都为单操作数指令，它们都能自动地修改 8087 的标志字，以反映指令执行之后寄存器的状态。表 10.8 列出了所有的数据传送类指令。

表 10.8 数据传送指令

实 数 传 送	FIST 存整数 FISTP 存整数并弹出堆栈
FLD 取实数 FST 存实数 FSTP 存实数并弹出堆栈 FXCH 寄存器交换	紧凑格式的 10 进制数传送
	FBLD 取紧凑格式的 10 进制数(BCD)
整 数 传 送	FBSTP 存紧凑格式的 10 进制数(BCD)并弹出堆栈
FILD 取整数	

1．FLD 源

取实数指令 FLD 将源操作数压入寄存器栈顶。具体过程是：先将堆栈顶指针减 1，然后把源操作数送到新的栈顶。这里的源操作数可以是堆栈中某个寄存器的内容，也可以是存贮器中的一个实型操作数。倘若源操作数的类型为长实数或短实数，那么指令 FLD 将自动地将其转换为临时实数格式，再装入栈顶。FLD ST(0)的作用是复写栈顶寄存器的内容。

2．FST 目标

存实数指令 FST 把栈顶寄存器的内容送至目标当中，其中的目标可以是堆栈中的其它寄存器，也可以是具有长实数或短实数类型的存贮器操作数。由于栈顶寄存器的内容(隐含的源操作数)总是以临时实数格式表示的，故在执行 FST 指令时，需要按照控制字中 RC 字段的规定，把源操作数有效字段舍入成目标操作数相应字段的宽度，并将其指数转换为目标格式所要求的宽度，还要对指数的偏移值进行修正。

3，FSTP 目标

这是一条存实数并弹出堆栈指令，它与指令 FST 的区别仅在于：它除了完成存实数操作之外，还要弹出栈顶的内容。即执行该指令之后，原栈顶寄存器已被标志为空，状态字 ST 字段的内容也被加 1，以指向新的栈顶。此外，FSTP 还能把寄存器中的内容以临时实数格式存放至内存当中，这一点是 FST 所做不到的。指令 FSTP ST(0)的作用为栈顶弹出，它不作其它任何数据传送。

4．FXCH//目标

寄存器交换指令 FXCH 将目标寄存器的内容与栈顶寄存器的内容互相交换。假如没有明确指定目标操作数，那就把 ST(1)用作目标操作数。8087 指令集中的许多指令都只是对栈顶内容进行操作，因此，为了充分有效地使用这样一些指令，FXCH 是一个很好的工具，它能够根

据指令的需要把操作数取至栈顶，且不再另占寄存器。例如，下面这段程序用于计算堆栈中第三个寄存器的平方根：

```
FXCH ST(3)
FSQRT
FXCH ST(3)
```

5．FILD 源

取整数指令 FILD 把源操作数从其 2 进制整数格式转换为临时实数格式，并将转换所得结果压入堆栈。其中的源操作数可以是整数字、短整数或长整数。若源操作数的所有位皆为 0，则把新栈顶标记为“0”，否则标记为“有效”。

6．FIST 目标

存整数指令 FIST 按照控制字中 RC 字段的规定把栈顶的内容舍入成整型数，并将经舍入所得到的结果送存到目标操作数当中。其中的目标操作数可被定义为整数字或短整数格式。

7．FISTP 目标

存整数并弹出堆栈指令 FISTP 的作用类似于指令 FIST，只是它除了完成数据传送任务之外，还弹出寄存器堆栈顶的内容，此外，该指令的目标操作数可以具有任何一种 2 进制整数类型，包括长整数。

8．FBLD 源

取紧凑格式的 10 进制数(BCD)指令 FBLD 将源操作数从 BCD 转换为临时实数，并将该结果压入堆栈。FBLD 的操作是精确的，它不带任何舍入误差地取进源操作数，源操作数的符号也被保留。但也应清楚，源操作数中的各个 10 进制数字都应在 0～9 的范围内，该指令不对无效数字(A～F)进行检验，因此，若源操作数中存在这种无效数字的编码时，所得到的结果将是无意义的。

9．FBSTP 目标

这是一条存数并弹出堆栈指令，它将栈顶的内容先转换为一个紧凑的 10 进制整数，然后把结果存放到存贮器的目标操作数中，再弹出栈顶寄存器。在进行数据类型之间的转换时，FBSTP 先在栈顶的非整数(临时实数)值上加上 0.5，然后舍去小数部分，再对该整数值 (2 进制)进行转换，化为 2—10 进制数，可见，该指令传送的是一个经四舍五入处理过的整数。不希望产生进位的用户可以在 FBSTP 指令之前，先用一条 FRNDINT 指令。

§10.3.2 比 较 指 令

比较指令常用于分析寄存器堆栈顶的内容与某个别的操作数之间的关系，这里的操作数可以是存贮器中的整型数或实型数，也可以是寄存器堆栈中其它的某个寄存器。比较、分析的结果可以从状态字的条件码中得到反映。比较指令又可以分成比较、测试和检验三种基本类型。为了便于对比较、测试或检验的结果进行检查，在比较操作之后可以利用 FSTSW 指令将状态码送到存贮器中保存起来，使用这种技巧可以实现程序的条件分支转移功能。在一次比较操作之后，立即保护状态字的另一个理由是：除了比较类指令之外，其它的一些 8087 指令也可能更新状态码，所以，上面的保护措施能够有助于保证状态字不致于在无意中被改动。

表 10.9 列出了 8087 的比较类指令：

表 10.9 8089 的比较类指令

FCOM	实数比较
FCOMP	实数比较并弹出堆栈
FCOMPP	实数比较并两次弹出堆栈
FICOM	整数比较
FICOMP	整数比较并弹出堆栈
FTST	测试
FXAM	检验

1．FCOM//源

实数比较指令 FCOM 把栈顶的数据与源操作数进行比较，其中，源操作数可以是堆栈中的某个寄存器，也可以是具有长/短实数格式的存贮器操作数。倘若不指明源操作数，那么就把栈顶内容(ST)与 ST(1)进行比较。在指令执行之后，条件码就反映出了比较的结果，对应关系如下：

C_3	C_0	关　　系
0	0	ST＞源操作数
0	1	ST＜源操作数
1	0	ST＝源操作数
1	1	ST 与源操作数的关系不定(源是 NAN 或 ∞)

2．FCOMP//源

实数比较并弹出堆栈指令 FCOMP 的作用与指令 FCOM 类似，只是该指令还外加一个弹出堆栈操作，这是 FCOM 所没有的。

3．FCOMPP

该指令的比较操作是在栈顶寄存器 ST 与下一个寄存器 ST(1)之间进行的，因此，不需要显式给出操作数。至于比较动作，该指令与 FCOM 也是类似的，不同点仅在于 FCOMPP 还要进行两次出栈操作，即把执行该指令之前位于寄存器堆栈顶的两个寄存器弹出堆栈。

4．FICOM 源

这是一个整数比较指令，它先把类型为整数字或短整数的源操作数化为临时实数，然后再把它与栈顶内容相比较。

5．FICOMP 源

整数比较并弹出堆栈指令 FICOMP 所执行的操作与指令FICOM 基本相同，区别只在于它还执行一次弹出栈顶动作。

6．FTST

测试指令 FTST 实际上是把栈顶寄存器的内容与 0 进行比较，从而达到"测试"栈顶的目的。

测试结果可用条件码表示如下：

C_3	C_0	比　较　结　果
0	0	ST＞0
0	1	ST＜0
1	0	ST＝0
1	1	不好对 ST 进行比较(因为它是一个 NAN 或 ∞)

7．FXAM

检验指令 FXAM 对栈顶寄存器的内容进行检查，通过设置条件码，指出栈顶内容是正是负，还指出该内容是 NAN、非规格化数、未规格化数、规格化的、0 或空的。这条指令在实际应用中是比较有用的，后面我们将用例子来说明这一点。FXAM 指令所产生的条件编码如下：

条件码				说明
C_3	C_2	C_1	C_0	
0	0	0	0	+unnormal(非规格化)
0	0	0	1	+NAN
0	0	1	0	−unnormal
0	0	1	1	−NAN
0	1	0	0	+normal
0	1	0	1	+∞
0	1	1	0	−normal
0	1	1	1	−∞
1	0	0	0	+0
1	0	0	1	Empty(空的)
1	0	1	0	−0
1	0	1	1	Empty
1	1	0	0	+denormal(未规格化)
1	1	0	1	Empty
1	1	1	0	−denormal
1	1	1	1	Empty

§10.3.3 算术运算指令

作为专用于数值运算的一个协处理器，8087 提供了更为有力、更为灵活的一些算术运算指令，这是由它的本质所决定的。8087 的算术运算指令除了包括基本的加、减、乘、除操作之外，还包括一般机器所没有的绝对值的计算、平方根的计算等指令，这些指令的执行速度比原始的除法操作还要快，因此，对于 8087 的程序员来讲，已不再需要为从算法中消除掉运行得“太慢”的平方根之类的计算而花费精力。表 10.10 列出了 8087 所提供的几条特殊的算术运算指令。表 10.11 则给出了 8087 的基本算术运算指令及其操作数组合格式。

表 10.10 特殊算术运算指令

FSQRT	求平方根
FSCALE	换算
FPREM	求部分余数
FRNDINT	舍入成整数
FXTRACT	分解指数和有效位
FABS	求绝对值
FCHS	改变符号

表 10.11 基本算术运算指令

指令格式	记忆格式	操作数格式目标，源	ASM-86 例子
传统堆栈	FOP	(ST(1),ST)	FADD
寄存器	FOP	ST(i),ST 或 ST,ST(i)	FSUB ST,ST(3)
寄存器弹出	FOPP	ST(i),ST	FMULP ST(2),ST
实数存贮器	FOP	(ST)短实数/长实数	FDIV AZIMUTH
整数存贮器	FIOP	(ST_1)字整数/短整数	FIDIV N_PULSES

注：括号( )中为隐含的操作数，它们在指令中并不出现.

OP=ADD　目标←目标+源
SUB　目标←目标−源
SUBR　目标←源−目标
MUL　目标←目标*源
DIV　目标←目标÷源
DIVR　目标←源÷目标

8087 的基本算术运算指令是为了便于开发效率更高的算法而设计的。除了一般计算机都具有的加、减、乘、除操作指令之外，8087 的基本算术指令还包括了两种逆向操作指令，这就使得减法和除法操作也变为对称的。利用这些指令，程序员就能够适当减少访问存贮器的次数，使运算过程尽量在寄存器堆栈中进行，这一方面使得 8087 的寄存器堆栈得到了最佳的利用，提高了算术运算指令的执行速度；另一方面，也使得 8087 占用局部总线的次数相对减少，从而提高了主机与 8087 之间的并行工作程度。

从表 10.11 中可以看出，各条指令及其操作数形式也是相当灵活的，具体表现在：

1．操作数既可以位于寄存器当中，也可以来自存贮器单元；

2．运算所得到的结果放到某个寄存器中；

3．操作数的类型是多种多样的，可以是临时实数、长实数、短实数、短整数、整数字，在进行计算时，8087 将自动地把它们转换为临时实数，然后才真正开始计算过程。

表 10.11 中列出的算术运算指令的五种基本形式可以与表下给出的六种操作相组合，形成具体的算术运算指令。传统堆栈形式可以使 8087 象一般的堆栈机器那样工作，在这种形式的指令中，只需要写出指令助记符，而不要求给出操作数，在执行这类指令时，NDP 从栈顶取出源操作数，从栈顶的下一个单元中取出目标操作数，从堆栈中弹出源/目标寄存器，执行操作，最后把运算结果压入堆栈，相当于用结果取代了原目标操作数寄存器中的内容；寄存器形式是传统堆栈形式的推广，这类指令把寄存器堆栈的栈顶用作其中的一个操作数，另一个操作数可由栈中的任一寄存器承当，运算结果总是送至目标寄存器当中；对于某些操作来说，栈顶寄存器仅仅是其中的一个源操作数，且在本次操作完成之后，它就不起其它作用了，因而它也就没有继续存在下去的必要，在这种情况下，就应该使用寄存器弹出形式的指令，这种形式的指令从栈顶取得源操作数，并执行一次“弹出栈顶”动作；除了上面介绍的三种基本形式之外，8087 的算术运算指令还包括两种存贮器形式，这两种形式的指令允许把存贮器中的某个实数或 2 进制整数用作源操作数，其中的存贮器操作数可以利用主机的任意一种寻址方式进行寻址，由于这两种形式的存在，8087 的算术运算也就更加灵活。对于那些不常用到的操作数来讲，为了节省高速的、有限的寄存器空间，一般总是把它们放在存贮器当中，这样就可以保证寄存器堆栈获得最佳利用，提高整个程序的执行速度。此外，由于主机寻址方式的灵活性，又使得算术运算指令可以方便、有效地运用于数组及其它数据结构。

下面，我们准备逐条地介绍一下 8087 的算术运算指令，先讨论六种基本操作，然后再介绍另外的七条算术运算指令。

1．加法指令

加法指令执行加法，它把源操作数与目标操作数相加，结果送至目标操作数。8087 的加法指令共有三种类型：实数相加、实数相加且栈顶弹出、整数相加。对应的指令格式为：

FADD//源/目标，源

FADDP 目标，源

FIADD 源

例如，指令

FADD ST,ST(0)

将使栈顶内容加倍。

2．减法指令

这里的减法代表一般的减法，它执行的动作是：从目标操作数中减去源操作数，并把差值送至目标操作数。减法指令也有三种类型，它们是：减实数、减实数并弹出栈顶、减整数，对应的指令格式为：

FSUB//源/目标，源

FSUBP 目标，源

FISUB 源

3．逆向减法指令

逆向减法指令与上述减法指令不同，尽管它也是执行减法操作，但它是从源操作数中减去目标操作数，差值也被送到目标操作数。三种逆向减法指令为：逆向减实数、逆向减实数并弹出栈顶、逆向整数减，对应的指令格式如下：

FSUBR//源/目标，源

FSUBRP 目标，源

FISUBR 源

4．乘法指令

乘法指令所执行的主要操作是：把源操作数和目标操作数相乘，并把乘积送到目标操作数当中。三种乘法指令是：乘实数、乘实数并弹出栈顶、整数乘法，对应的指令格式为：

FMUL//源/目标，源

FMULP 目标，源

FIMUL 源

例如，下面这条指令将使栈顶内容自乘，即原栈顶内容的平方：

FMUL ST,ST(0)

5．除法指令

除法指令（包括除实数、除实数并弹出堆栈、整数除指令）将目标操作数除以源操作数，并把商送到目标操作数当中。除法指令的三种形式如下：

FDIV//源/目标，源

FDIVP 目标，源

FIDIV 源

6．反向除法指令

上面介绍的是一般的(正常的)除法指令，它与反向除法指令不同，反向除法指令是用源操作数除以目标操作数，商也被送到目标操作数当中．反向除法指令也有三种方式：反向除实数、反向除实数并弹出堆栈、反向整数除，指令格式分别为：

FDIVR//源/目标，源

FDIVRP 目标，源

FIDIVR 源

7．FSQRT

FSQRT 是一条计算平方根的指令，它求的是寄存器堆栈顶内容的平方根，其结果将取代栈顶寄存器的原始值．需要指出一点，在 8087 中，规定 $\sqrt{-0}=0$。

8．FSCALE

指令 FSCALE 把 ST(1)的内容当作一个整数值，加到栈顶寄存器(ST)中的指数部分上。也就是说，指令 FSCALE 的作用是：

$$ST \leftarrow ST * 2^{ST(1)}$$

由此可见，指令 FSCALE 提供了一种以 2 的整数次幂为因子的快速乘法 (ST(1)＞0) 或除法 (ST(1)＜0)，这对于向量元素的变换是特别有用的。

但应注意，临时实数的指数字段也仅由 15 位组成，因此，为了使本指令能够正确地得到执行，指令 FSCALE 假设放在 ST(1)中的“比例因子” X 是一个满足条件 $(-2^{15} \leqslant X \leqslant 2^{15})$ 的整数。假如X不是一个整数，但它落在上述范围内且其绝对值大于 1，那么 FSCALE 就利用截尾舍入方式将X化为整数，然后将此整数值加到 ST 的指数上。但是，倘若 ST(1)的内容超出了上述范围，或者其绝对值小于 1 的话，那么，执行该指令之后得到的将是一个无定义的结果。为了确保指令的正确执行，我们提倡把比例因子的类型定为整数字。

9．FPREM

指令 FPREM 是一条求部分余数的指令，执行该指令时，栈顶内容被栈顶下一个单元的内容作模除，即 ST(1)中的内容是模数，所得到的结果(部分余数)放在栈顶寄存器当中。

指令 FPREM 是通过连续地执行减法操作而实现其功能的，因此，假如两个操作数 (ST 和 ST(1))的值之间的差别太大，若要获得精确的余数，就将耗费大量的执行时间，由于该指令的执行时间较长，故需考虑许多其它因素(如在指令执行期间发出中断)，因而，把该指令安排在一个由软件控制的循环中反复执行．程序通过对 ST 和 ST(1) 进行比较来判定取模操作是否应该结束，即如果 ST＞ST(1)，那么就必须再次执行 FPREM；若 ST＝ST(1)，那么余数等于 0；若 ST＜ST(1)，那么余数就在 ST 当中。

指令 FPREM 的一个重要用途就是调整周期性超越函数的自变量(操作数)，使其落在这些函数指令所允许的自变量取值范围之内。例如，对于求正切函数值的指令来讲，它要求其操作数（幅角）在 0 到 $\pi/4$ 之间，为了使指令能够正确执行，可以用 $\pi/4$ 作模数，利用指令 FPREM 把原幅角调整到上述范围(0 到 $\pi/4$)内，然后再使用 FPTAN 指令计算正切函数值。

指令 FPREM 不会引起精度事故，它所形成的运算结果是精确的。结果(余数)的符号和原被除数的符号相同(ST 的符号)。

10．FRNDINT

FRNDINT 是一条取整指令，它根据控制字中 RC 字段的规定，把栈顶寄存器的内容舍入

成整数．例如，假设 ST 的内容是 155.615（用 10 进制表示），如果控制字中 RC 字段的内容是 01（向下舍入方式），则 FRNDINT 的结果就为 155；如果控制字中把舍入方式规定为向上舍入（RC = 10）或向最近值舍入（RC = 00），那么 ST 中的值就将变成 156。

11．FXTRACT

指令 FXTRACT 把栈顶寄存器的内容分成两个数，这两个数分别表示操作数的指数和有效字段的实际值，操作数的指数实值取代栈中的原操作数（原栈顶），而有效字段的真值则被压入堆栈，成为新栈顶的内容。也就是说，在执行了 FXTRACT 指令之后，ST（即新的栈顶）中含有以临时实数格式表示的原操作数（原栈顶的内容）有效字段的值，其符号位和原操作数的符号位相同，指数字段的内容是真 0（即偏移值 3FFFH），至于有效字段，其内容就是原操作数有效字段的内容。而 ST(1)（原栈顶）则含有以临时实数格式表示的原操作数的真指数。为了便于理解，我们举两个例子来说明 FXTRACT 的作用。假设 ST 中含有一个数，其真指数是 +4，即指数字段是 4003H，在执行完指令 FXTRACT 之后，ST(1) 将含有实数 +4.0，即其符号位为0，指数字段的内容为4001H（真指数是 +2），有效字段的内容为1△00……00B。换句话说，ST(1) 的值为 $1.0*2^2=4$。假设 ST 中操作数的真指数不是 +4，而是 −7，即指数字段的值是 3FF8H，那么，在执行了 FXTRACT 指令之后，ST(1) 将含有实数值 −7.0，即其符号位为1，指数字段的值是4001H（真指数为 +2），有效字段的内容为1.1100…00B，这样，ST(1) 的值就等于 $-1.11B*2^2=-7.0$。无论原操作数的值是多少，在执行了指令 FXTRACT 之后，ST 的符号位和有效字段均与原操作数的符号位及有效字段相同，而其指数字段的内容则为 3FFFH（真指数是 0）。

FXTRACT 指令同 FBSTP 指令一起使用时，就可以比较方便地实现临时实数到 10 进制数的转换，这对于打印或显示结果是很有用处的，本章后面的例子将说明这一用途。此外，FXTRACT 指令对于程序的调试也相当有用，因为利用该指令之后，可以分别对实数的指数部分和尾数部分进行检查。

12．FABS

这是一条求绝对值的指令，它所执行的动作仅仅是把栈顶单元的符号置为正号（0）。

13．FCHS

指令 FCHS 改变栈顶寄存器中的符号，即若 ST 的符号为正，执行该指令之后，ST 的符号就变为负，反之亦类似。

§10.3.4 超越函数计算指令

8087 提供了五条超越函数计算指令，这几条指令是经过仔细研究所精心选择出来的。利用这几条指令，就能够完成所有的三角函数、反三角函数、双曲线函数、反双曲函数、对数函数和指数函数的主要计算任务，即完成了超越函数计算过程中最费时间的那些部分的操作，从而大大提高了所有这些超越函数的计算速度。8087 的五条超越函数计算指令如表 10.12 所示。

这些指令都是在寄存器堆栈顶实现它们的操作功能的，操作数来自栈顶的一个或两个寄存器单元，函数值（运算结果）也将被送到堆栈当中。在执行这些指令之前，可以先用软件把其操作数的幅值调整到对应指令所能接受的范围内，然后再计算其函数值；在计算过程完成之后，若有必要还需把结果调整到和原幅值相对应的值域上。所有超越函数指令都假定它们的操作数是有效的，且在规定的范围内。对于一个超越函数的操作数来讲，有效一定是规格化

表 10.12 超越函数计算指令

FPTAN	部分正切
FPATAN	部分反正切
F2XM1	计算 2^X-1
FYL2X	计算 $Y\cdot\log_2 X$
FYL2XP1	计算 $Y\cdot\log_2(X+1)$

的，至于那些未规格化的、非规格化的、NAN 及无定义的操作数则都是无效的，假如超越函数指令的某个操作数不满足要求，即若它是一个无效操作数或超出了规定范围的话，那么对应的指令就将产生一个无定义的结果，而并不指出是一次事故。因此，作为程序设计者，他应该在执行超越函数指令之前，尽力保证其操作数是有效的且位于规定的取值范围之内。前面在介绍 FPREM 指令的时候，我们已经指出过指令 FPREM 在周期函数的计算中可以用于此种目的。

1. FPTAN (Partial tangent)

指令 FPTAN 用于计算函数 $Y/X=TAN(\theta)$，其中的操作数 θ 位于栈顶寄存器当中，它必须满足条件 $0\leqslant\theta<\frac{\pi}{4}$。运算的结果是一个比值 Y/X，Y 取代栈中的 θ，X 则压入堆栈，成为新栈顶的内容。之所以把该指令的结果取为比值，而不是单一的函数值，是因为考虑到这样可以优化或有利于其它三角函数值的计算，显然，利用 X 和 Y 可以很方便地求得 θ 的正弦值、余弦值、反正切值……。

2. FPATAN (Partial arctangent)

指令 FPATAN 用于计算函数 $\theta=ARCTAN(Y/X)$，其中，X 从栈顶寄存器中取出，Y 从 ST(1) 中取来，且它们必须满足关系：$0\leqslant Y<X<\infty$。在执行完该指令时，原来存放 X 及 Y 的寄存器都被弹出堆栈，结果 θ 成为新栈顶的内容，实际上，θ 取代了原操作数 Y。

3. F2XM1

指令 F2XM1 用于计算函数 $Y=2X-1$，其中，X 取自栈顶，它必须满足条件：$0\leqslant X\leqslant 0.5$，结果 Y 将取代栈顶的 X。

即使操作数 X 很接近于 0，该指令也能形成相当精确的结果。倘若所需要计算的是 2X，而不是 2X－1，那么就只要把指令 F2XM1 所求得的结果加上 1 即可。尽管这条指令仅仅能够计算 2 的指数值，然而，利用下面这些众所周知的公式，就可以实现其它指数函数的计算。

$$10X=2X^{\cdot\log_2 10}$$

$$eX=2X^{\cdot\log_2 e}$$

$$YX=2X^{\cdot\log_2 Y}$$

后面我们还将看到，8087 还有这样一些指令，利用它们可以装入常数 $\log_2 10$、$\log_2 e$，此外，指令 FYL2X 又可以用来计算 $Y\cdot\log_2 X$ 这种函数的值，由此可见，只要适当地把这些指令与指令 F2XM1 组合起来，就能够方便、有效地实现 10X、eX 及 YX 这样一些函数值的计算。还应看到，把指令 F2XM1 用于计算双曲函数值也是有利可图的。

4. FYL2X

指令 FYL2X 用于计算函数 $Z=Y\cdot\log_2 X$，这里的 X 取自栈顶寄存器，Y 位于寄存器 ST(1) 当中，两个操作数所必须满足的条件是：$0<X<\infty$，$-\infty<Y<+\infty$。该指令执行之后，原栈顶已被弹出，结果 Z 将成为新的栈顶内容，即它取代了操作数 Y。利用对数的底数变换公式，可以把任意的不是以 2 为底数的对数函数变换成为以 2 为底的对数函数，从而使得指令

FYL2X 可用于实现任意对数函数值的计算。

5．FYL2XP1

指令 FYL2XP1 用于计算函数 $Z=Y\cdot\log_2(X+1)$，其中 X 取自栈顶，它必须满足条件 $0<|X|<1-\frac{\sqrt{2}}{2}$；Y 来自 ST(1)，其取值范围是 $-\infty<Y<+\infty$。该指令也将引起一次弹出堆栈操作，其运算结果 Z 将取代操作数 Y，成为新的栈顶数值。

当“自变量” X 非常接近于 0 时，该指令能够比 FYL2X 更为精确地计算出它的对数值。例如，假设需要计算 $1+\varepsilon$ 的对数值，其中 $\varepsilon\ll1$，那么，对于指令 FYL2XP1 来讲，只要求输入 ε，而不是 FYL2X 所要求的 $1+\varepsilon$，显然，这就使得“输入”的有效位数更多；从而提高运算的精度。

§10.3.5 常 数 指 令

常数指令的作用就是把常数压入堆栈。所选择的几个常数是在程序当中有可能经常出现的，这些常数具有与临时实数格式相同的精度，即差不多可以精确到 19 个 10 进制数字。大家知道，临时实数在存贮器中要占用 10 个字节，而一条常数指令仅仅占用两个存贮器字节，可见利用常数指令可以节省存贮空间。实际上，利用常数指令还提高了程序的执行速度，简化了程序设计。七条常数指令列于下表当中：

表 10.13 常数指令

FLDZ	装入 +0.0
FLD1	装入 +1.0
FLDPI	装入 π
FLDL2T	装入 $\log_2 10$
FLDL2E	装入 $\log_2 e$
FLDLG2	装入 $\log_{10} 2$
FLDLN2	装入 $\log_e 2$

1．FLDZ (load zero)

取 0 指令 FLDZ 将 +0.0 装入(压入)堆栈。

2．FLD1 (load one)

取 1 指令 FLD1 将 +1.0 压入堆栈。

3．FLDPI (load π)

取 π 指令 FLDPI 将 π 压入堆栈。

4．FLDL2T

指令 FLDL2T 将常数 $\log_2 10$ 压入栈堆。

5．FLDL2E

指令 FLDL2E 将常数值 $\log_2 e$ 压入堆栈。

6．FLDLG2

指令 FLDLG2 将常数值 $\log_{10} 2$ 压入堆栈。

7．FLDLN2

指令 FLDLN2 将常数值 $\log_e 2$(ln2)压入堆栈。

§10.3.6 处理器控制指令

处理器控制类指令原则上是用于控制系统的活动，而不是用于计算。利用它们，可以对处理器的初始化、异常事件的处理及任务的转换等等活动实施控制。下表列出了 8087 的处理器控制类指令。

表 10.14 处理器控制指令

FINIT/FNINIT	处理机初始化
FDISI/FNDISI	禁止中断
FENI/FNENI	允许中断
FLDCW	装入控制字
FSTCW/FNSTCW	保存控制字
FSTSW/FNSTSW	保存状态字
FCLEX/FNCLEX	清除事故
FSTENV/FNSTENV	保存环境
FLDENV	装入环境
FSAVE/FNSAVE	保护状态
FRSTOR	恢复状态
FINCSTP	堆栈指针递增
FDECSTP	堆栈指针递减
FFREE	使寄存器空闲
FNOP	空操作
FWAIT	CPU 等待

从上表中可以看出，不少处理器控制指令都具有两种形式，即有两种指令助记符。同一指令的两种形式之间的区别就在于是否需要把 CPU 的 WAIT 指令用作为该指令的前缀。顺便在这里提一下，在 8087 的汇编语言当中，多数指令的前面都附有一条 WAIT 指令，其目的就是使 8087 与主机能够同步地进行工作。例如，指令 FSQRT 的宏代码定义可能是：

```
FSQRT   macro
        WAIT
        ESC 0FH,DX
ENDM
```

对于处理器控制指令 FENI/FNENI（允许中断）来讲，尽管两者的作用相同，然而两者对应的宏代码定义是有区别的。我们可以把 FENI 的宏代码定义为：

```
FENI    macro
        WAIT
        FNENI
ENDM
```

而把 FNENI 的宏代码定义为：

```
FNENI   macro
        ESC 1CH,AX
ENDM
```

可见，FENI 仅仅比 FNENI 多执行了一条 WAIT 指令，其它指令对应的情况也与此类似。所

有这些指令中的 WAIT 都起同步作用，这将在后面详细介绍。

对 8087 的绝大多数指令来讲，其宏代码定义中都包含一条 WAIT 指令，只有表 10.14 中的几条处理器控制指令是不带 WAIT 的。这种“不等待”的形式通常用于一些很“关键”的地方，在这样一些地方，利用 WAIT 指令很可能使 CPU 进入无穷等待状态，即发生死锁现象。就一般情况而言，当 CPU 中断被禁止，而 8087 有可能产生中断时，应该使用这些指令的“不等待”形式；反之，即若 CPU 允许中断，则应使用这些指令的“等待”形式。倘若要求保证做到在 8087 的 NEU 所在执行的操作结束后，才执行处理器控制指令，那么，这些处理器控制指令都应使用“等待”形式。

1．FINIT/FNINIT (initialize processor)

初始化处理器指令 FINIT/FNINIT 除了不会影响 8087 和 CPU 同步地取指令之外，其操作功能是与硬件信号 RESET 等效的。对于处理器初始化之后的情况，将在下一节中介绍。

2．FDISI/FNDISI (disable interrupts)

FDISI/FNDISI 指令是一条禁止中断指令，其作用是将 8087 控制字的中断允许/屏蔽位 (IEM 字段)置为“1”，表示禁止 8087 发出中断请求。

3．FENI/FNENI (enable interrupts)

指令 FENI/FNENI 与上一条指令相反，它是一条允许中断指令，执行该指令后，控制字中的中断允许/屏蔽位将被清零，从而允许 8087 发出中断申请信号。

4．FLDCW 源

指令 FLDCW 的作用是装入处理器控制字，即用由源操作数所规定的那个字取代当前的处理器控制字。该指令一般用于建立或改变 8087 的操作方式。在使用该指令的时候，需要注意这样一种情况：如果在执行 FLDCW 之前，状态字中的某个事故位已被置为“1”，表示曾经发生过对应的异常事件，而就在这种情况下，处理器 8087 执行一条 FLDCW 指令，所装入的控制字既清除了中断允许/屏蔽位，也清除了相应的事故屏蔽位，这样，在执行下一条指令之前，8087 就将产生一个中断请求信号。由于这一现象的存在，所以我们建议在改变 8087 的操作方式(例如，用指令 FLDCW)之前，先清除掉所有的事故标志，然后再装入新的处理器控制字。

5．FSTCW/FNSTCW 目标

这是一条保存处理器控制字的指令，其作用就是把 8087 的当前控制字写至目标操作数所规定的存贮器单元之中。

6．FSTSW/FNSTSW 目标

这是一条保存 8087 状态字的指令，它把 8087 当前状态字的内容写到目标操作数所指定的存贮器单元。该指令有这样一些用途：

(1) 接在比较指令或 FPREM 指令的后面，实现条件分支转移，此时应该利用 FSTSW；

(2) 确定 8087 是否处于“忙”状态，此时应利用 FNSTSW；

(3) 在不使用中断的环境中，利用 FSTSW 指令可以实现事故处理程序的调用。

实际上，8087 或 CPU 都是通过查询由指令 FSTSW 保存在存贮器中的状态，来完成上面这些功能的。

7．FCLEX/FNCLEX

指令 FCLEX/FNCLEX 清除状态字中的所有事故标志、中断申请标志和“忙”标志，即执行该指令之后，8087 的 INT 和 BUSY 信号都将变为无效(为 0)。在从事故处理程序返回到被

中断的计算程序之前，必须执行这条指令，否则，又一次中断请求就将发生，从而引起死循环。

8．FSAVE/FNSAVE 目标

指令 FSAVE/FNSAVE 将 8087 的全部状态环境(环境加寄存器堆栈的内容)保存到由目标操作数所指定的存贮器区域当中。这些状态环境共占 94 个字节的存贮空间，在存贮器中的格式如图 10.16 所示，该指令的目标操作数通常是指向 CPU 堆栈的一个地址变量，即一般把 8087 的状态环境保存在 CPU 的栈段当中。

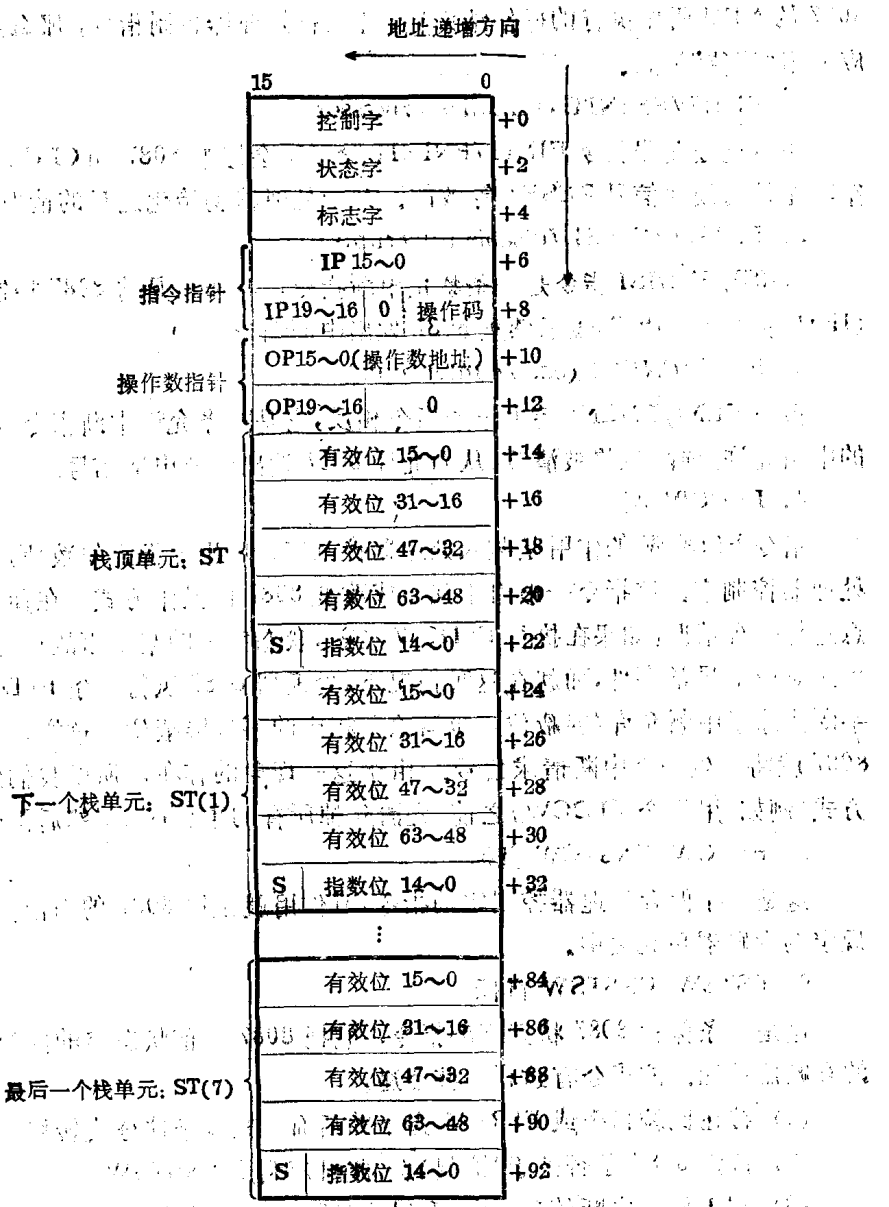


图 10.16 分配给 FSAVE/FRSTOR 所保存的环境状态的存贮器区域格式

当使用 FSAVE 时，它当然要等到 8087 的现行工作完成之后才开始执行；但如果使用的是 FNSAVE，且在译码 FNSAVE 时，8087 的 NEU 正在执行一条指令，那么，8087 也将在 NEU 当前的指令正常地结束或遇到一次未被屏蔽的事故时，才开始执行指令 FNSAVE，保护状态环境。因此，这条指令所保护的信息反映了前面的指令执行完成时 8087 的状态。FSAVE/

FNSAVE 除了保护 8087 的状态环境之外，它还将对 8087 进行初始化处理，即在完成状态的保护之后，接着执行与指令 FINIT/FNINIT 相同的操作，初始化 8087。由此可见，每当某段程序需要保护 8087 的当前状态，且需要为执行新的程序而初始化 8087 时，就应该使用指令 FSAVE/FNSAVE。在出现下面某种情况时，可以使用这条指令：

(1) 操作系统需要进行一次任务调度，即中止当前任务的继续执行，而把控制转移到另一个任务，或曰让另一任务投入运行；

(2) 中断处理程序需要使用 8087，这也类似于任务调度；

(3) 应用程序希望把"被清洗过"的 8087 用于执行一段子程序。

9. FRSTOR 源

指令 FRSTOR 所起的作用与 FSAVE/FNSAVE 相反，它是一条状态恢复指令，即把源操作数所指向的存贮区(94 个字节)的内容装入 8087 的相应寄存器当中。一般来说，存贮区中的这些信息是由先前的 FSAVE/FNSAVE 指令所保护起来的，它反映了前面某个时候 8087 的状态，因此，应避免使用可能会修改这些内容的指令。当指令 FRSTOR 执行结束时，8087 就又进入了一个新的状态环境，但要注意，这一状态有可能会使 8087 立即发出一次中断请求。

10. FSTENV/FNSTENV 目标

指令 FSTENV/FNSTENV 也起着保护 8087 状态的作用，不过它所保护的是 8087 的所谓基本状态(环境)，即仅仅保护 8087 的控制字、状态字、标志字以及事故指针的内容，而不保护寄存器堆栈，这些基本状态都将保存到由目标操作数所指定的存贮区中，通常是保存到 CPU 的栈上面。图 10.17 说明了环境信息在存贮器中的存放格式。

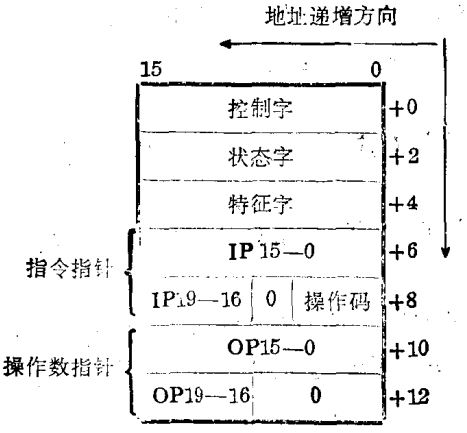


图 10.17 FSTENV/FNSTENV 的存贮器分配格式

在异常事件(事故)处理程序当中，往往需要使用指令 FSTENV/FNSTENV，因为利用该指令可以把 8087 的环境——包括事故指针——送到存贮器中，所以事故处理程序通过访问存贮器就能够确定引起事故的指令及操作数，从而实现对事故的处理。指令 FSTENV/FNSTENV 在保护完了现场之后，就把处理器内部的所有事故屏蔽位置位，但它不影响中断允许/屏蔽位的状态。与 FNSAVE 指令类似，如果在译码 FNSTENV 的同时，NEU 正在执行另外某条指令的话，那么，8087 就把这条 FNSTENV 指令排上队，只有在 NEU 执行完当时的那条指令时，才开始执行 FNSTENV，保护 8087 的现场。可见，指令 FNSTENV 所保护起来的信息反映的是前一指令执行完之后 8087 的现场。

指令 FSTENV/FNSTENV 必须在任何别的 8087 指令被译码之前完成其操作功能，因此，

对于跟在 FSTENV/FNSTENV 后面的任何一个 8087 指令来讲，除非它带有由汇编程序所产生的 WAIT，否则就应在其前面加上一条 FWAIT 指令。在中断被禁止处，若要使用该指令来保护环境，则应该使用"不等待"形式 FNSTENV。

11. FLDENV 源

指令 FLDENV 的作用与上一条指令相反，它是一条装入环境指令，即从源操作数所指示的存贮区中读数，把它们装入各自的寄存器(对应于图 10.17 给出的格式)，从而恢复 8087 的环境现场。读进的数据区一般应由先前的某条环境保护指令 FSTENV/FNSTENV 所建立。在指令 FLDENV 之后，可以紧跟上不访问环境映象区的 CPU 指令，但后续的 8087 指令之前，则应加上 FWAIT 指令，除非该 8087 指令本身隐含地带有 WAIT。在装入环境时，也应该注意这一现象的发生：倘若给 8087 装入的是一个含有未被屏蔽的事故的环境映象，且控制字中的 IEM = 0，那么就会使得 8087 立即发出中断请求。

12. FINCSTP

指令 FINCSTP 所执行的操作仅仅是把状态字中的栈顶指针 (ST 字段的内容)加 1。它不改变寄存器及标志字的内容，也不进行数据传送，由于它不会把先前栈顶寄存器相应的标志位置为"空"，可见该指令的操作不同于弹出堆栈。若该指令执行前的 ST 等于 7，那么执行 FINCSTP 之后的 ST 就将成为 0。

13. FDECSTP

指令 FDECSTP 与指令 FINCSTP 类似，都是对状态字中的栈顶指针 ST 进行处理，不同点只是指令 FDECSTP 是把 ST 的值减 1。执行该指令之后，标志字及寄存器的内容都不发生变化，也没有进行任何数据传输，可见，指令 FDECSTP 也不同于一般的压栈操作。当 ST = 0 时，执行指令 FDECSTP 将使 ST 的内容变为 7。

14. FFREE 目标

指令 FFREE 把目标寄存器对应的标志置为"空"，寄存器中的内容不受该指令的影响。

15. FNOP

这是一条空操作指令，它所执行的动作只是把栈顶送到栈顶，即相当于执行一条

```
FST ST,ST(0)
```

指令。可见，指令 FNOP 除了占有一定的存贮器空间及执行时间之外，不起任何作用，即空操作。但在设计程序时，这种空操作指令还是有用的，尤其适用于对程序的调试、修改过程。

16. FWAIT

指令 FWAIT 实际上不属于 8087，FWAIT 只是 CPU WAIT 指令的另一个助记符。其宏代码定义为：

```
FWAIT  macro
       WAIT
endm
```

这条指令是为了保证 CPU 和 8087 同步地进行工作而设置的，无论什么时候，只要程序员希望在这一时刻使 CPU 和 8087 获得同步，就应该使用一条 FWAIT 指令，它能够保证 8087 在结束当前指令的执行之前，不会对下一条指令进行译码处理，从而在执行完 FWAIT 指令时，可以确保 CPU 及 8087 都已经完成了前面的操作，即获得了同步。此外，在 8087 的指令结束之前，我们自然会希望 CPU 指令不要访问该 8087 指令正在读写的存贮器操作数，特别是当

8087 指令所执行的是存数操作时，这一点就尤为重要，而利用 FWAIT 指令，也能达到这方面的要求。下面这段程序就说明了利用 FWAIT 来迫使 CPU 指令等待 8087 完成其当前指令所采用的方法：

```
    ⋮
FNSTSW  STATUS
FWAIT                  (等待指令 FNSTSW 执行完毕)
MOV     AX,STATUS
    ⋮
```

§10.3.7 8087 指令系统小结

前面我们已经逐条地对 8087 的所有指令进行了比较详细的介绍，但对于如何利用它们来编制程序之类的问题还没作讨论。为了给读者一个 8087 程序设计的印象，下面给出一个简单的例子。假设有三个已被定义过的变量 x、y、z，实际上就是三个存贮器地址，x 中的值是一个短实数(x是一个短实数变量)，y 是一个整数字变量，z 是一个长实数变量，并设我们希望计算：

$$x=\frac{(x^2+y^2)}{x+y}+\sqrt{(z-y)}$$

那么，下面这段程序就可以解决这个问题：

```
FLD     LONG z   ; Load z into 8087 stack (把 z 中的内容取至 8087 的寄存器堆栈
                 ; 顶)
FISUB   WORD y   ; Subtract integer y from z, leave ST (执行(z-y)，结果留在栈
                 ; 顶)
FSQRT            ; Take square root of z-y (求√(z-y)的值)
FLD     SHORT x  ; Load short-real x (把短实数 x 取到栈顶上)
FMUL    ST,ST(0) ; Multiply by itself to get x-squared (计算 x²)
FILD    WORD y   ; Load word-integer y (取整数字 y)
FMUL    ST,ST(0) ; Get y-squared (求 y 的平方)
FADD             ; Add x-squared to y-squared (求 x²+y²)
FLD     SHORT x  ; Load short-real x again
FIADD   WORD y   ; Add word-integer y to x (求 x+y)
FDIV             ; Divide (x+y) into (x²+y²) (求(x²+y²)/(x+y))
FADD             ; Add result to √(z-y) (求得(x²+y²)/(x+y)+√(z-y))
FST     SHORT x  ; Store result back as short-real x (把结果以短实数格式存放到
                 ; 存贮器单元 x 当中)
```

程序段中的 LONG、WORD、SHORT 都是汇编语言中的伪指令，它们在程序中用来指明变量的类型，详细情况这里就不准备介绍了。

有关指令表的注解：

mod 和 r/m 字段用于寻址存贮器操作数，其含义与 8086/8088 中的相应字段完全一致。

表 10.15 8087的指令表

数据传送类	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
FLD：取				
整型/实型存贮器到ST(0)	ESCAPE MF 1	MOD 0 0 0 R/M	(DISP-LO)	(DISP-HI)
长整型存贮器到ST(0)	ESCAPE 1 1 1	MOD 1 0 1 R/M	(DISP-LO)	(DISP-HI)
临时实型存贮器到ST(0)	ESCAPE 0 1 1	MOD 1 0 1 R/M	(DISP-LO)	(DISP-HI)
BCD存贮器到ST(0)	ESCAPE 1 1 1	MOD 1 0 0 R/M	(DISP-LO)	(DISP-HI)
ST(i)到ST(0)	ESCAPE 0 0 1	1 1 0 0 0 ST(i)		
FST：存				
ST(0)到整型/实型存贮器	ESCAPE MF 1	MOD 0 1 0 R/M	(DISP-LO)	(DISP-HI)
ST(0)到ST(i)	ESCAPE 1 0 1	1 1 0 1 0 ST(i)		
FSTP：存并弹出堆栈				
ST(0)到整型/实型存贮器	ESCAPE MF 1	MOD 0 1 1 R/M	(DISP-LO)	(DISP-HI)
ST(0)到长整型存贮器	ESCAPE 1 1 1	MOD 1 1 1 R/M	(DISP-LO)	(DISP-HI)
ST(0)到临时实型存贮器	ESCAPE 0 1 1	MOD 1 1 1 R/M	(DISP-LO)	(DISP-HI)
ST(0)到BCD存贮器	ESCAPE 1 1 1	MOD 1 1 0 R/M	(DISP-LO)	(DISP-HI)
ST(0)到ST(i)	ESCAPE 1 0 1	1 1 0 1 1 ST(i)		
FXCH：ST(i)与ST(0)内容交换	ESCAPE 0 0 1	1 1 0 0 1 ST(i)		
比较类				
FCOM：比较				
整型/实型存贮器与ST(0)比较	ESCAPE MF 0	MOD 0 1 0 R/M	(DISP-LO)	(DISP-HI)
ST(i)与ST(0)比较	ESCAPE 0 0 0	1 1 0 1 0 ST(i)		
FCOMP：比较且出栈				
整型/实型存贮器与ST(0)比较	ESCAPE MF 0	MOD 0 1 1 R/M	(DISP-LO)	(DISP-HI)
ST(i)与ST(0)比较	ESCAPE 0 0 0	1 1 0 1 1 ST(i)		
FCOMPP：ST(1)与ST(0)比较且弹出两次	ESCAPE 1 1 0	1 1 0 1 1 0 0 1		
FTST：测试ST(0)	ESCAPE 0 0 1	1 1 1 0 0 1 0 0		
FXAM：检验ST(0)	ESCAPE 0 0 1	1 1 1 0 0 1 0 1		
算术运算类				
FADD：加法				
整型/实型存贮器与ST(0)相加	ESCAPE MF 0	MOD 0 0 0 R/M	(DISP-LO)	(DISP-HI)
ST(i)与ST(0)相加	ESCAPE d P 0	1 1 0 0 0 ST(i)		
FSUB：减法				
整型/实型存贮器与ST(0)相减	ESCAPE MF 0	MOD 1 0 R R/M	(DISP-LO)	(DISP-HI)
ST(i)与ST(0)相减	ESCAPE d P 0	1 1 1 0 R R/M		

续表

	76543210	76543210	76543210	76543210
FMUL：乘法				
整型/实型存贮器与ST(0)相乘	ESCAPE MF 0	MOD0 0 1 R/M	(DISP-LO)	(DISP-HI)
ST(i)与ST(0)相乘	ESCAPE d P 0	1 1 0 0 1 R/M		
FDIV：除法				
整型/实型存贮器与ST(0)相除	ESCAPE MF 0	MOD1 1 R R/M	(DISP-LO)	(DISP-HI)
ST(i)与ST(0)相除	ESCAPE d P 0	1 1 1 1 R R/M		
FSQRT：求ST(0)的平方根	ESCAPE 0 0 1	1 1 1 1 1 0 1 0		
FSCALE：ST←ST*$2^{ST(2)}$	ESCAPE 0 0 1	1 1 1 1 1 1 0 1		
FPREM：ST(0)÷ST(1)的余数	ESCAPE 0 0 1	1 1 1 1 1 0 0 0		
FRNDINT：把ST(0)舍入成整数	ESCAPE 0 0 1	1 1 1 1 1 1 0 0		
FXTRACT：分解成指数和有效位	ESCAPE 0 0 1	1 1 1 1 0 1 0 0		
FABS：ST(0)的绝对值	ESCAPE 0 0 1	1 1 1 0 0 0 0 1		
FCHS：改变ST(0)的符号	ESCAPE 0 0 1	1 1 1 0 0 0 0 0		
超越函数计算类				
FPTAN：ST(0)的正切值	ESCAPE 0 0 1	1 1 1 1 0 0 1 0		
FPATAN：ST(0)/ST(1)的反正切值	ESCAPE 0 0 1	1 1 1 1 0 0 1 1		
F2XM1：$2^{ST(2)}-1$	ESCAPE 0 0 1	1 1 1 1 0 0 0 0		
FYL2X：ST(1)·$\log_2$[ST(0)]	ESCAPE 0 0 1	1 1 1 1 0 0 0 1		
FYL2XP1：ST(1)·$\log_2$[ST(0)+1]	ESCAPE 0 0 1	1 1 1 1 1 0 0 1		
常数指令				
FLDZ：ST(0)←0.0	ESCAPE 0 0 1	1 1 1 0 1 1 1 0		
FLD1：ST(0)←1.0	ESCAPE 0 0 1	1 1 1 0 1 0 0 0		
FLDPI：ST(0)←π	ESCAPE 0 0 1	1 1 1 0 1 0 1 1		
FLDL2T：ST(0)←$\log_2 10$	ESCAPE 0 0 1	1 1 1 0 1 0 0 1		
FLD2E：ST(0)←$\log_2 e$	ESCAPE 0 0 1	1 1 1 0 1 0 1 0		
FLDLG2：ST(0)←$\log_{10} 2$	ESCAPE 0 0 1	1 1 1 0 1 1 0 0		
FLDLN2：ST(0)←$\log_e 2$	ESCAPE 0 0 1	1 1 1 0 1 1 0 1		
处理器控制类				
FINIT：初始化8087	ESCAPE 0 1 1	1 1 1 0 0 0 1 1		
FENI：允许中断	ESCAPE 0 1 1	1 1 1 0 0 0 0 0		
FDISI：屏蔽中断	ESCAPE 0 1 1	1 1 1 0 0 0 0 1		

续表

	76543210	76543210	76543210	76543210
FLDCW：取控制字	ESCAPE 001	MOD101 R/M	(DISP-LO)	(DISP-HI)
FSTCW：存控制字	ESCAPE 001	MOD111 R/M	(DISP-LO)	(DISP-HI)
STSW：存状态字	ESCAPE 101	MOD111 R/M	(DISP-LO)	(DISP-HI)
FCLEX：清除事故	ESCAPE 011	11100010		
FSTENV：存环境	ESCAPE 001	MOD110 R/M	(DISP-LO)	(DISP-HI)
FLDENV：装入环境	ESCAPE 001	MOD100 R/M	(DISP-LO)	(DISP-HI)
FSAVE：保护状态	ESCAPE 101	MOD110 R/M	(DISP-LO)	(DISP-HI)
FRSTOR：恢复状态	ESCAPE 101	MOD100 R/M	(DISP-LO)	(DISP-HI)
FINCSTP：增堆栈指针	ESCAPE 001	11110111		
FDECSTP：减堆栈指针	ESCAPE 001	11110110		
FFREE：使 ST(i)空闲	ESCAPE 101	11000ST(i)		
FNOP：无操作	ESCAPE 001	11010000		
FWAIT：CPU 等待 NPX	10011011			

注：MF 代表存贮器操作数的格式，编码如下：

MF	存贮器操作数类型
00	32 位实数
01	32 位整数
10	64 位实数
11	64 位整数

ST(0)代表当前栈顶寄存器

ST(i)代表栈顶下面第 i 个寄存器

d 字段用于确定目标操作数，即 d 为 0 时表示 ST(0)为目标操作数，d 为 1 时 ST(i)为目标操作数。

P 字段的内容确定了指令是否需要完成弹出堆栈操作，为 0 时不弹，等于 1 时则需弹出栈顶 ST(0)

R 为反向字段，即确定运算进行的方向，

当 R 等于 0 时：目标(OP)源

当 R 等于 1 时：源(OP)目标

§10.4 数值处理机的体系结构

在前面几节当中，我们已经比较详细地介绍了数值协处理器 8087。而作为 8086/8088 的一个辅助处理机，8087 又是如何与其主机相连、如何并行工作的呢？即数值处理机究竟是

怎样形成的呢？这些问题将在下面两节中得到解答，其中，本节着重介绍数值处理机的体系结构(硬件)，下一节则侧重于数值处理机的程序设计方面(软件)。

§10.4.1 8087 与 8086/8088 之间的连接

8087 作为 8086/8088 CPU 的一个协作处理器，能够加到工作在最大模式的任何 iAPX86/1X 或 88/1X 系统中，从而形成数值处理机 iAPX 86/2x 或 88/2x。数值处理机 iAPX 86/20 (或 iAPX 88/20) 中局部总线的连接情况如图 10.18 所示，从图中我们能够看到 8087 与其主机 8086/8088 之间的连线情况。实际上，8087 可以与工作在最大模式的主机共用多重地址/

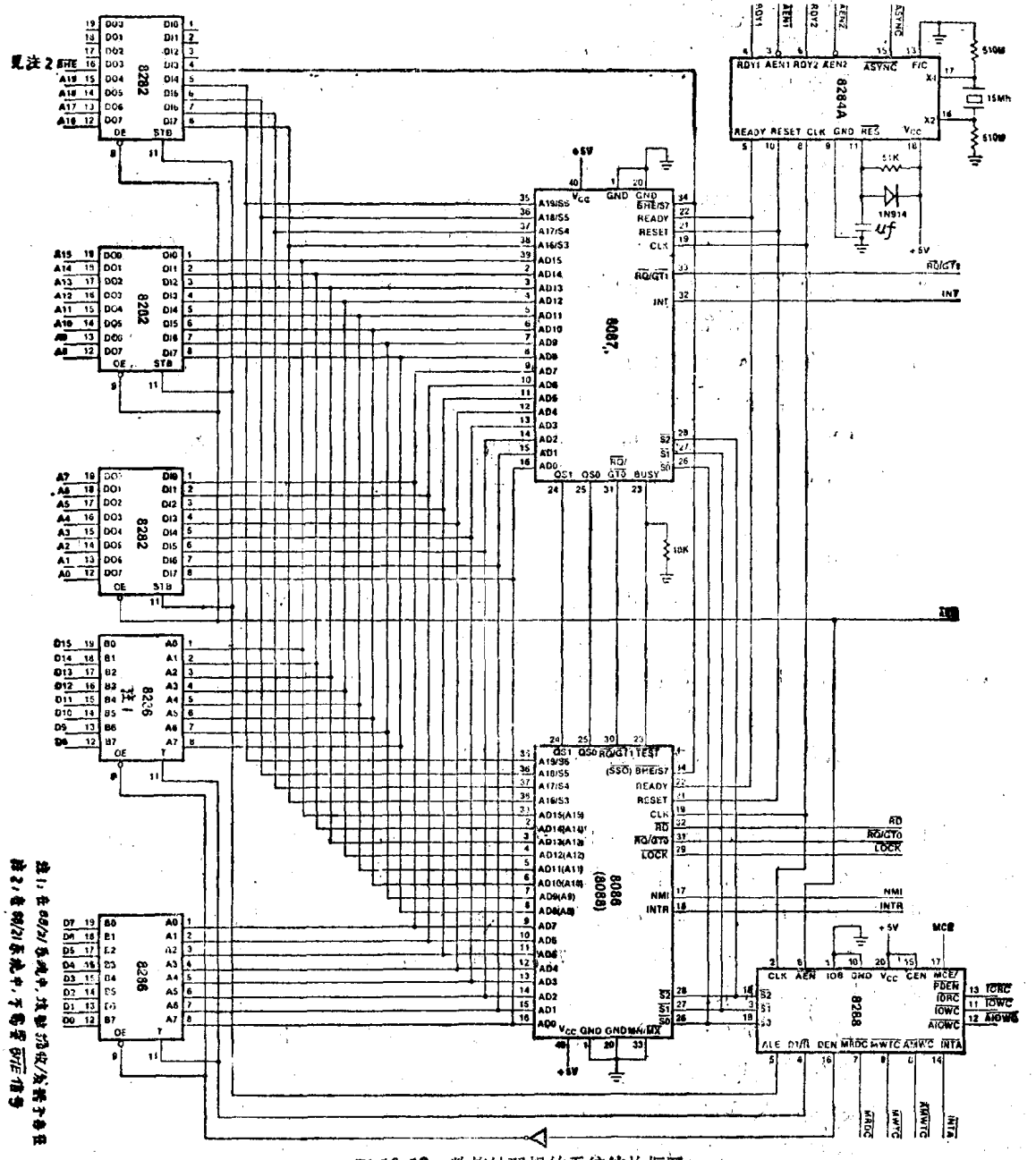


图 10.18 数值处理机的系统结构框图

数据总线、状态线、队列状态线以及 CLK、READY 和 RESET 信号，8087 中两个很有用的信号 BUSY 和 INT 用于通知主机当前的 8087 状态，CPU 的状态线(S0,S1,S2)和队列状态线(QS0, QS1) 使得 8087 能够在不增加 CPU 开销的情况下，与 CPU 同步地监督和译码指令。8087 一经启动就能与主机并行地、独立地工作，为了重新获得同步，8087 用 BUSY 信号通知 CPU 当时 8087 是否正在执行指令，若 8087 正在执行操作，那么就把 BUSY 置为高电平，否则 BUSY 为低电平，这样，CPU 就可以利用 WAIT 指令来判定 8087 是否可以执行下面一条指令，从图 10.18 中可以看到，8087 的 BUSY 引脚是与主机的 TEST 相接的，当系统中没有加上 8087 时，在原接 BUSY 信号的地方有一个 10K 欧姆的下拉电阻，这就使得主机总是看到一个处于"不忙" 状态的 8087。当 8087 发现错误或异常事件时，它就中断 CPU，8087 的中断请求信号一般是经过 8259A 中断控制器送给主机的。

在设计这样的系统时，我们可以先给 8087 留一个 40 引脚的插座，根据具体应用的需要，确定是否需要在系统中加上 8087，当要求系统具有较高的数值处理能力时，就把一片 8087 插到留给它的那个空插座中，这就相当于把 8087 加到系统当中去了。倘若系统中没有加8087，而某个程序又企图执行数值指令，那么主机就把这些数值指令当作空操作指令 NOP 处理，而并不把此作为出错对待。数值处理机中的所有处理器(8086/8088，8087，8089)都使用相同的总线接口部件。顺便提一下，8087 是 8086 系列中的一个协处理器，它除了可以加到以 8086/8088 为基础的系统中之外，还可以加到 CPU 为 80186 或 80286 的系统当中。

借助于软件能够检查系统中是否存在 8087，采用的方法为先初始化 8087，然后保存其控制字，根据控制字的内容(0 或非 0)就可以确定 8087 存在与否，因为若存在 8087，则在初始化之后，其控制字的内容不等于 0(=3FFH)。程序如下所示：

```
ESC   28, BX          (若 8087 存在，就相当于执行指令 FNINIT，初始化 8087)
XOR   AX, AX          (这两条指令一方面起延时的作用，让 8087 在此期间实现初始
                       化动作，另一方面，它把单元 control 的内容置为 0)
MOV   control, AX
ESC   15, control     (若 8087 存在，则执行指令 FNSTCW control，将控制字存到
                       control 中，否则，空操作)
OR    AX,control      (若 8087 存在，那么单元 control 中的内容为 3FFH，否则等于
                       0)
JZ    no—8087         (若 8087 不存在，则跳转到 no—8087)
:
```

除了上面的指令 FNINIT 能够初始化 8087 之外，FINIT 及 FSAVE/FNSAVE 指令也能对 8087 进行初始化处理，这些都是用指令即从软件上实现初始化动作的，除此之外，还可以从硬件上着手，即利用 RESET 信号来开始初始化过程，当 8087 发现 RESET 信号变为有效时，它就终止当前的一切活动，并对自身进行初始化处理。不管采用哪种方式，经过初始化处理后的 8087 状态如表 10.16 所示。

当然，硬件初始化与软件初始化之间也有区别。硬件初始化(RESET 信号)期间，8087 还要识别其主机类型(8086 或 8088)，并且开始跟踪 CPU 的取指操作；而软件初始化不会影响 8087 对 CPU 的跟踪状况，在此过程中，也无需识别主机的类型。从表 10.16 中看到，初始化处理不影响寄存器及事故指针的内容，它们都具有不定值，但是，由于在初始化处理之后，状态

表 10.16 初始化后的 8087 状态

字　　段	值	说　　明
控制字		
无穷大控制	0	发射的
舍入控制	00	舍入到最接近的值
精度控制	11	64 位
中断允许屏蔽	1	禁止中断
事故屏蔽	111111	屏蔽所有事故
状态字		
忙	0	不忙
条件码	????	(不定)
栈顶	000	堆栈是空的
中断申请	0	无中断
事故标志	000000	无事故
标志字		
标志位	11	空的
寄存器	无　值	不被修改
事故指示器		
指令编码	无　值	不被修改
指令地址	无　值	不被修改
操作数地址	无　值	不被修改

字中的 ST 字段的内容为 0，所有寄存器对应的标志位都为 11，表示这些寄存器都是空的，可见，初始化之后的寄存器堆栈实际上是空的，因此，一般可以认为寄存器的内容被初始化过程所"破坏"。

在 RESET 之后，8087 通过检查 $\overline{\text{BHE}}$/S7 的状态，就能够确定其主机类型。从图 10.18 中能够看到，8087 的 $\overline{\text{BHE}}$/S7 引脚是与 CPU 的第 34 号引脚相连的，若 CPU 是 8086，则第 34 号引脚为 $\overline{\text{BHE}}$/S7，若 CPU 是 8088，它就代表 SSO 信号。在访问存贮器时，8088 总是把 SSO 保持为 1，而 8086 在访问位于偶地址的存贮器字或位于奇地址的字节时，则发出 $\overline{\text{BHE}}$ 信号（为 0）。在接到 RESET 信号之后，主机就从专用存贮单元 FFFF0H 处开始执行指令，在这一点上，如果主机是 8086，它就执行一次读字操作，此时，8087 也就从 $\overline{\text{BHE}}$/S7 引脚上接到一个低电平信号；而当主机为 8088 时，它就执行一次读字节操作，这时，8087 从 $\overline{\text{BHE}}$/S7 引脚上则收到一个高电平信号(SSO)。这样，8087 就可以确定主机的类型了。

数值处理机(NDP)所执行的指令都是 CPU 与 8087 共同活动的结果，CPU 和 8087 都有它们自己的寄存器、指令系统及一些特殊功能部件，这些也是它们能够并行操作的基础与前提。由于它们之间的这种并行工作潜力以及 8087 巨大的数值处理能力，使得数值处理机的数值运算能力比原来的 8086/8088 系统(没加 8087)提高了将近一百倍，表 10.17 给出了一些典型数值指令的执行时间。CPU 负责控制整个程序的执行过程，而 8087 则利用它的协处理器接口来识别和完成数值指令。当 8087 驱动总线周期时，S3、S4 及 S6 为高电平，而 S5 为低电平。当 8087 在监督 CPU 总线周期时，它可以通过检查 S6 的状态将 8086/8088 的活动与局部 I/O 处理器或任何别的总线主设备区别开来，这是因为只有 8086/8088 才能将 S6 变成低电平。

表 10.17 数值运算的速度对比

指令类型		处理机类型	
		iAPX 86/20(5MHz)(8087+8086)	iAPX 86/10(5MHz)(8086)
		执行时间(单位：μs)	
浮点运算指令	加/减	17	1,600
	单精度乘法	19	1,600
	双精度乘法	27	2,100
	除	39	3,200
	比较	9	1,300
	取双精度数	10	1,700
	存双精度数	21	1,200
	求平方根	36	19,600
	求正切	90	13,000
	指数运算	100	17,100
18 位 BCD 运算指令	加/减	127	12,040
	乘	297	22,990
	除	323	26,560
	比较	150	20,250

§10.4.2 iAPX86/88 的协处理器接口

考虑到某些专用处理机或专用硬件能以最快的速度、最简单的结构、最低的价格实现某种功能，从而引进了协处理器的设计思想。通常是由于价格的原因，使得我们不能把所有的功能都集中在一个通用硬件上实现，当然，实际上也很少需要系统具有所有的功能，这就要求提供一些用于特定环境的支持部件。

通用微处理器 8086 或 8088 的协处理器接口(coprocessor interface)提供了一种简单、高效的增加专用硬件的环境。协处理器接口很简单地与主机的局部地址/数据、状态、时钟、READY、RESET、TEST 及请求/同意信号相连，如图 10.18 所示，使用这种直接接到主机的局部总线上的方法，就使得协处理器能够访问主机的所有存贮器及 I/O 资源。由于协处理器的硬件是为了解决某些特殊问题而专门设计的，所以，它一般能以与 CPU 相同的规模及价格，提供比 CPU 高得多的解决这些特殊问题的速度，即大大提高了系统的性能/价格比。此外，协处理器并不依赖于系统的具体结构，这是因为它是接到主机的局部总线上的，而局部总线的定时及功能又是固定不变的，这也决定了使用协处理器的方便性和通用性。

协处理器接口使得这些专用硬件(专用协处理器)都表现为主机的某个部分，主机利用一些特殊指令来控制它们，当主机碰到这些特殊指令时，它就跟协处理器一道识别它们，并且协同工作，完成所需要的动作。对协处理器来讲，它所需要的往往不仅仅是一个指令操作码和一个开始执行信号，它还要求得到更多的信息，例如许多指令都要求操作数，因此，主机的协处理器接口还能根据需要，从存贮器中取一个数值或给协处理器一个它所要访问的存贮区域，而协处理器又可以使用主机的所有寻址方式来寻址存贮器操作数。

在协处理器开始执行它自己的操作之后，主机可以继续运行程序，即在同一时刻，两个处理器都在执行各自的任务。一般来讲，除非协处理器要访问存贮器操作数或 I/O 操作数，在其

它情况下，协处理器的并行操作不影响主机的运行。即使在协处理器访问存贮器或 I/O 操作数，而主机把局部总线让给协处理器的时候，主机仍然能够执行其内部指令队列中的指令，但是，倘若主机此时也需要使用局部总线(如主机指令也要求访问存贮器操作数)，那么主机就得暂停当时的操作，直等到协处理器释放总线时为止。除了挪用存贮器周期之外，协处理器的所有其它活动对主机来讲都是透明的。我们把主机与协处理器之间的这种并行运行方式叫做并发执行(concurrent execution)。很显然，指令并发执行的速度要比顺序执行快，并发执行方式的系统性能也要更高一些。

程序的并发执行确实要比顺序执行快，但它也带来了一些问题，其中最主要的就是同步问题，即在并发执行的系统当中，程序必须提供主机与协处理器之间的同步功能，这是因为主机和协处理器往往需要从对方那里获得某些信息。这里的同步包括这样两种情况，一种是主机等待协处理器完成其现行操作，另一种是协处理器在等待主机完成其当前操作。由于主机执行程序，具有象跳转这样的程序控制指令，因而同步的任务就让它来承担。为了满足这一需要，主机提供了一条专门用于同步的指令，即 WAIT 指令。指令 WAIT 通过检查主机 TEST 引脚的状态，确定是该等待(TEST = 1)还是继续执行后继指令(TEST = 0)，而协处理器也是通过这条引脚告诉主机它当前是否"忙"着，从图 10.18 当中可以看到，协处理器的 BUSY 引脚是与主机的 TEST 引脚相连的，当协处理器在执行操作时，就将把 BUSY 置为高电平，此信号通过连线就传到了主机的 TEST 端，而当主机执行 WAIT 指令时，若 TEST 为高电平时，它就暂停程序的执行(等待)，一旦 TEST 变为低电平(协处理器完成了其现行操作)，主机就恢复程序的运行，执行 WAIT 下面的那条指令。

现在，简单地讨论一下协处理器为 8087 时的指令同步情况。对一条典型的数值指令来讲，CPU 所看到的只是一个 ESC 指令，而指令的解释及执行则主要由 8087 完成，可见，主机对数值指令的处理是相当简单的，它仅仅是译码一条 ESC 指令，最多也不过是计算一次操作数地址和读出一个操作数字，因此，主机能够远在 8087 之前完成对该数值指令的处理。例如，8087 完成一次平方根的计算大约需用 180 个时钟周期，而 CPU 对同一指令所需进行的处理只要用两个时钟周期。主机在完成了 ESC 指令之后，就将译码、执行后面的指令，前面曾经说过，8087 当中有一个与其主机相同的指令队列，当主机使其指令队列发生变化时，8087 也将相应地修改它的指令队列，这一取指令的同步功能是由 8087 的控制部件(CU)完成的，8087 对所有的 CPU 指令都不作任何处理，实际上 8087 所能处理的也仅仅是一些属于它的数值指令。如果 CPU 所做的工作不影响 8087，那么，在 8087 执行数值指令的同时，CPU 就可以继续执行下面的一串指令，8087 的控制部件 CU 除了跟踪这些 CPU 指令(调整指令队列)之外，不执行任何别的操作。以上主要是讲主机和 8087 取指令过程中的同步问题，除了取指令需要同步之外，指令在执行过程中也要求一定的同步，粗略地讲，前者是由硬件保障的，即前一同步是由控制部件 CU 提供的；而后者则主要得由软件来控制，即在程序当中考虑同步要求。执行过程中的同步又分下面两种情况：

1．在 8087 的 NEU 正忙于执行前一条数值指令时，它一定不能开始执行下一条 8087 指令；

2．如果 8087 正在执行的是一条需要访问存贮器操作数的指令，那么在不能肯定 8087 已经完成那个单元的访问之前，CPU 不应该访问那个存贮器单元。

对于这两种情况的同步要求，只要在软件当中，借助于主机的 WAIT 指令就可以了。

WAIT 指令能够保证，在 8087 完成其当前指令之前，CPU 不执行后面的指令。有关这方面的内容将在下一节中详细讨论。

在具有协处理器的系统当中，整个程序是由主机进行控制的，协处理器的运行也是由主机的一些特殊指令启动的，这些特殊指令就是 8086/8088 的换码指令(ESC)。当主机碰到一条 ESC 指令时，就让协处理器去完成指令所规定的动作，主机的协处理器接口要求协处理器能够确定主机何时遇到了一条 ESC 指令，一旦主机开始执行一条新指令，协处理器就必须检查该指令是否是一个 ESC 指令。由于只有主机才能取出这些指令并执行它们，所以协处理器必须不停顿地监测主机处理指令的过程。

主机能够在执行某个指令之前的某个时刻取进该指令，这一点也是 8086/8088 微处理器指令队列的一个特色，指令队列使得主机能在局部总线空闲时预取指令，从而使得主机指令可以比较快地得到执行。主机不给外界指出它在执行的是哪条指令(这一点不同于普通计算机，因为以往的多数计算机中的程序计数器 PC 可以起这方面的作用)，而只是当它取指令或译码、执行一个操作码字节时，主机才真正指出它正在处理的指令，因此，为了确定主机从它的队列中取出的究竟是哪条指令，协处理器就必须保持一个与协处理器相同的指令队列。根据主机的类型及转移指令的目标地址确定取指令是以字节为单位还是以字为单位，当主机的指令队列已满时，就不再预取指令。主机每次从指令队列中取出一个字节去译码和运行，如果发生跳转动作，那么指令队列就将被清除，即成为一个空队列。协处理器通过监测主机的总线状态、队列状态及数据总线，就可以跟踪主机的上述活动。表 10.18 给出了总线状态信号和队列状态信号的编码情况。

表 10.18 总线状态及队列状态的编码

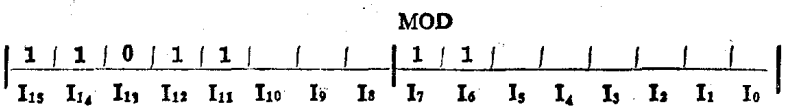
$\overline{S2}$	$\overline{S1}$	$\overline{S0}$	总线周期	QS1	QS0	主机活动	协处理器活动
0	0	0	中断响应	0	0	无	无队列操作
0	0	1	读 I/O 转接口	0	1	从队列中取出指令的第一个字节	译码操作码字节
0	1	0	写 I/O 转接口	1	0	队列已空	队列已空
0	1	1	暂停	1	1	从队列中取出指令的下一个字节	删去一个字节或当它为 ESC 指令的第二个字节时，对之进行译码
1	0	0	取指令				
1	0	1	读存贮器数据				
1	1	0	写存贮器数据				
1	1	1	空闲				

在许多系统当中，主机并不是唯一的能取指令的总线主设备，例如 Intel 8089 IOP 在执行存放在系统存贮器中的通道程序时，也可能要利用局部总线来取指令，此时，8089 所产生的总线状态信号(S0、S1 和 S2)与主机在取指令时所产生的状态信号相同，显然，协处理器不应该把这些指令当作主机的预取指令而送入其指令队列，状态信号 S6 提供了解决这个问题的手段。通过检查 S6 的状态，就能够确定局部总线是否由主机所占用，这是因为当主机是局部总线主时，S6 在存贮器周期的 T_2、T_3 阶段取 0 值(低电平)；而在其它的任一总线主设备取指令周期的 T_2 和 T_3 期间，S6 总是 1 (高电平)。既然如此，当 S6 = 1 时，协处理器就可以忽略局部总线上的活动。

为了识别换码指令 ESC，协处理器必须对主机所执行的所有指令进行检查。当主机从其指令队列中取出一个指令字节时，协处理器也将完成同样的操作。队列状态信号确定了主机何时在分析一个操作码字节，在主机分析操作码字节的同时，协处理器也将检查从内部指令队

列中取出的是否为换码指令 ESC 的操作码，若不是换码指令，协处理器就忽略掉该指令，即不对它进行任何处理。队列状态信号(11)使得协处理器在不了解主机指令及寻址方式的情况下，可以跟踪主机的指令队列。

所有换码指令 ESC 的开始五位均为 11011，它们有两种基本形式，一种为不访问存贮器的形式，其格式为：



指令中有九位可以用来规定协处理器所应完成的操作。换码指令的另一种基本形式为访问存贮器的 ESC 指令，其格式如图 10.19 所示

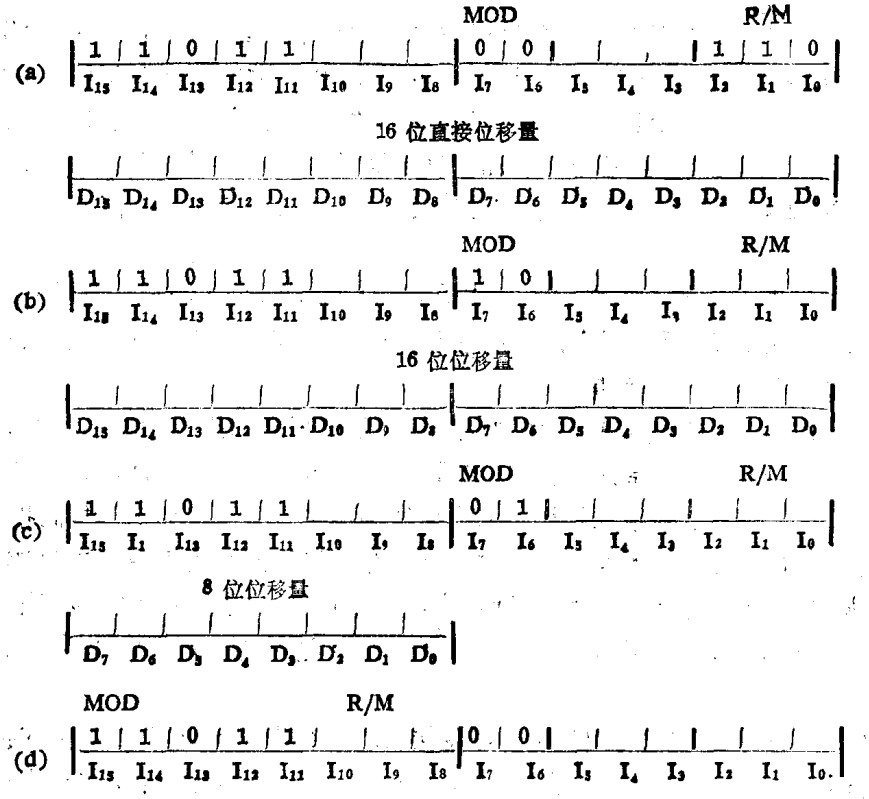


图 10.19 访问存贮器的换码指令格式

这种指令允许主机使用任意一种寻址方式来给协处理器指出一个存贮器操作数，从图中可见，此时，只有六位可用来规定协处理器应对存贮器操作数作何处理。从上面的讨论中可以看出，换码指令共有 576 条($2^9+2^6=576$)，可见，利用这些指令，可以使得协处理器完成许多不同的操作。当然，对一个协处理器来讲，它也许不能识别所有可能的换码指令，解决这一问题的方法也很简单，协处理器忽略掉它不能识别的那些 ESC 指令，即它不对这些指令进行任何处理。

当主机碰到换码指令时，它要么不做任何事情，要么计算一个存贮器单元的有效地址并从该单元中读取一个字，但 CPU 并不真的接收它读出的这个字。换码指令除了使主机的 IP 继续前进之外，不改变主机中任何别的寄存器的内容。因此，当系统中不存在协处理器，或当该换码指令是协处理器所不能识别的指令时，换码指令对主机的作用就跟空操作指令(NOP)一

样。实际上，主机为协处理器的活动所作出的贡献最多也只不过是计算一个存贮器地址，并从该存贮器单元中读出一个字而已。设置访问存贮器形式的换码指令的目的也有两个，即指定一个存贮器操作数，以及从存贮器中取出一个字送到协处理器。

倘若协处理器只需要从存贮器中读出16位或更小的数据的话，那么，协处理器的设计就简单多了，因为在这一前提下，即便是对于那些需要访问存贮器的换码指令，协处理器也只要在存贮器读周期的T_3结束时，读取并锁存出现在数据总线上的数值就可以了。这也就意味着，主机已经为协处理器完成了所有的访问存贮器的任务，而协处理器根本就不需要成为局部总线的主设备，去读/写别的信息。但是，实际上的协处理器常常需要把信息写到存贮器中，或者需要处理长于一个字的数值。这时，协处理器就必须保存存贮器的地址，并且要求占用局部总线，即有成为局部总线的主设备之必要。因此，在实际的系统当中，当主机执行换码指令(访问存贮器的ESC指令)时，它将在读存贮器周期的T_1间，把存贮器操作数的20位实际地址放到地址/数据总线上，此时，协处理器就能够接收并锁存这个地址，然后利用这个实际地址去访问所有的其它存贮器操作数字节(读存贮器操作数指令，操作数的第一个字已由主机读出)，或利用这一地址把结果送到存贮器当中(写存贮器操作数指令)。不管协处理器是否占用总线，在指令形式为访问存贮器的时候，它都必须能够确定执行换码指令时由主机所完成的存贮器读操作，即确定CPU的所谓"假读"周期，其实，只要满足下面两个条件，就能判定是这种存贮器读操作：

1．由主机执行的换码指令中第二个字节的MOD字段的值为00、01或10；

2．这是在主机碰到该ESC指令之后所进行的第一个读存贮器周期，特别是总线状态信号S2、S1、S0分别为1、0、1，而S6是0。

当主机在计算存贮器地址及读存贮器数据时，协处理器必须跟踪主机的指令队列，这是通过后面在队列状态引脚上的一系列"取下一字节"状态命令(QS0 = 1，QS1 = 1)实现的。

协处理器必须了解主机的总线特征，因为总线类型决定了主机如何为换码指令读取一个存贮器操作数字。如果主机为8088，那么它就将从存贮器的两个连续单元中完成两次读字节操作，即每次读一个字节；但若主机为8086，它就可能执行一次读字操作(该存贮器单元的地址为偶数)，也可能需要执行两次读字节操作(奇地址)，这是由读周期中T_1期间的AD_0所决定的。

综上所述，主机的换码指令ESC、协处理器接口和等待指令WAIT为主机的扩充(增加一些专用协处理器)提供了甚为方便的手段。8087就是一种专用协处理器，在程序员看来，数值处理机中的8087扩充了主机的结构，8087的硬件和软件非常经济地使系统性能得到了很大的提高。

下面着重讨论8087是如何使用8086/8088的协处理器接口这一问题。

8086/8088的换码指令ESC共提供了576种操作码，其中，有64个(2^6)需要访问存贮器的操作码，有512(2^9)个不访问存贮器操作数的操作码。8087共占用了这些操作码中的463个，即用掉了57个访问存贮器形式的操作码，和406个不访问存贮器形式的操作码。图10.20列出了没被8087使用的所有ESC指令。

设计师们可能想设计一些其它的协处理器，例如专用于进行文本编辑或图象处理的协处理器，在这种情况下，图10.20中所给出的这些操作码就显得相当宝贵了，因为这些协处理器不应该使用已经分配给8087的那些ESC指令的操作码，以免引起二义性或多义性问题，否

1	1	0	1	1				1	1						
I_{15}	I_{14}	I_{13}	I_{12}	I_{11}	I_{10}	I_9	I_8	I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0

I_{10}	I_9	I_8	I_5	I_4	I_3	I_2	I_1	I_0	可用的代码数
0	0	1	0	1	0	0	0	1	1
0	0	1	0	1	0	0	1	—	2
0	0	1	0	1	0	1	—	—	4
0	0	1	1	0	0	0	1	—	2
0	0	1	1	0	0	1	1	—	2
0	0	1	1	0	1	1	1	1	1
0	0	1	1	1	0	1	0	1	1
0	0	1	1	1	1	0	1	1	1
0	0	1	1	1	1	1	1	—	2
0	1	1	1	0	0	1	0	1	1
0	1	1	1	0	0	1	1	—	2
0	1	1	1	0	1	—	—	—	8
0	1	1	1	1	—	—	—	—	16
1	0	1	1	—	—	—	—	—	32
1	1	1	1	0	0	0	0	1	1
1	1	1	1	0	0	0	1	0	1
1	1	1	1	0	0	1	—	—	4
1	1	1	1	0	1	—	—	—	8
1	1	1	1	1	—	—	—	—	16
									计 105

可作它用的没引用存贮器的换码指令

								MOD						R/M	
1	1	0	1	1											
I_{15}	I_{14}	I_{13}	I_{12}	I_{11}	I_{10}	I_9	I_8	I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0

I10	I9	I8	I5	I4	I3
0	0	1	0	0	1
0	1	1	0	0	1
0	1	1	1	0	0
0	1	1	1	1	0
1	0	1	0	0	1
1	0	1	1	0	1
1	1	1	0	0	1

可作它用的引用存贮器的换码指令

图 10.20　8087 没用到的换码指令操作码

则，就不能确定某些指令是该由 8087 执行还是该由其它协处理器执行．通过前面对 8086/8088 协处理器接口的介绍，可以看出我们也能够根据各自的实用需要，设计出自己想要的协处理器，只是在把它与 8087 一起使用时，要受到下面两个方面的限制：

1．屏蔽掉 8087 的所有错误．这是因为对于那些不属于 8087 的 ESC 指令操作码，8087 也将修改其操作码寄存器及指令地址寄存器的内容，若这些 ESC 指令的形式是访问存贮器的，那么，8087 还将修改其操作数地址寄存器中的内容．而这样一些改变就使得原先定义的事故处理程序不能正确得到执行了．

2．倘若这些协处理器提供"忙"信号 BUSY，那么，就应该将所有协处理器的 BUSY 信号

"或"起来，接到主机的 TEST 引脚上。

当然，我们设计的这些协处理器可能会和 Intel 将要提供的协处理器发生矛盾，尤其是在操作码的选择方面，这种冲突更难避免。

8087 的存贮器操作数具有七种不同的格式，其中有六种都比一个字长，但是，细心一点可以发现这一现象，即它们的字节长度均为偶数，主机是通过这些数据最低字的地址来寻址其存贮器操作数的。如果某条访问存贮器的换码指令要求进行取数操作，那么，当主机执行该指令时，总要读出此 8087 存贮器操作数的低位字，8087 将把由主机所发出的操作数地址和读出的数据字保存起来，为了读取该操作数的其余部分，8087 就必须占用总线，在 8087 获得了局部总线之后，它就将前面保存起来的 20 位实际地址递增，以寻址操作数的其余部分。但当换码指令要求 8087 执行写存贮器操作时，尽管 8087 也要保存主机放到地址总线上的地址，然而它却不接收由主机所读出的那个数据字，当然，8087 最后也将获得总线，以完成连续的写操作。

8087 要与 8086 或 8088 主机一道工作，主机的类型决定了局部总线的宽度，8087 也将根据实际情况自动调整对数据总线的使用。当给主机和 8087 发出一个清除 (RESET) 信号时，8087 就将确定其主机的类型。在总清之后的第一个存贮器周期中，8087 将检查 34 号引脚的状态，若使用的是 8086 主机，34 号引脚就为 $\overline{BHE}$ 信号(低电平)，在此读存贮器字的周期中，它为低电平，故局部总线的宽度为 16 位。而对 8088 主机来说，34 号引脚代表 SSO 信号，它在第一个存贮器周期的 T_1 期间为高电平。可见，利用这个引脚，8087 就可以确定它应该使用哪种数据宽度，以便与主机局部总线的宽度相匹配。数据总线的宽度以及操作数在存贮器中的位置对 8087 的指令没有影响，受影响的只是指令的执行时间和所要执行的存贮器周期数。同一段数值处理程序在 86/2x 或 88/2x 上的运行结果总是一样的，其结果也不受操作数位置的影响，但是，存贮器操作数的字节定位能够影响在 86/2x 上运行的程序的执行速度，因为若字操作数或任何数值操作数的开始地址为奇数，那么，就需要较多的存贮器周期来完成操作数的存/取，这些额外的存贮周期降低了系统的性能，在 86/2x 系统当中，为了使访问那些奇定位操作数所需要的额外周期数达到最小，也做了一些努力，其方法是，8087 首先完成一次字节操作，然后进行一系列的字操作，最后再执行一次字节操作。当然，若规定数值操作数的地址为偶数，则将使 86/2x 的性能达到最佳状态。由于 88/2x 所执行的总是字节操作，所以，它不存在这样的问题。

§10.4.3 应该考虑的几个问题

在把 8087 接入系统时，必须作出两个抉择。第一就是要确定如何把所有局部总线上的主设备的 RQ/GT 信号线连接起来，因为这将影响到对局部总线请求的响应时间；另一个抉择就是确定如何连接中断，因为这将对中断的响应时间以及用户中断处理子程序的编写产生影响。这些引脚的连接方法与硬件设计者以及程序员都有关系，因此，必须设法弄清这些问题。

根据系统结构的不同，我们把 RQ/GT 的连接方式分成三个大类：86/20 或 88/20、86/21 或 88/21、86/22 或 88/22。8089 的远程操作不受 8087 RQ/GT 连接方式的影响。下面分别就上述三个大类的 RQ/GT 连接方式进行讨论：

1. iAPX 86/20、88/20

对 iAPX 86/20(8086 + 8087)或 88/20(8088 + 8087) 而言，只要将 8087 的 RQ/GT_0 引脚

与主机的 RQ/GT_1 引脚接起来就行了。参见图 10.18。

2 iAPX 86/21、88/21

在 iAPX 86/21 (8086+8087+8089) 或 88/21 中，将 8087 的 RQ/GT_0 接到主机的 RQ/GT_1，将 8089 的 RQ/GT 连到 8087 的 RQ/GT_1。如图 10.21 所示。

从图 10.21 中可以看到，8087 位于 8089 到主机的 RQ/GT 通路上面，它对 8089 从主机获得总线控制权的最大等待时间稍微有点影响，8087 使主机完成总线释放操作所需要的时间增加了两个时钟的延迟，这种延迟是由 8087 的 RQ/GT_1 上的竞争协议引起的。系统约定：接在 8087 RQ/GT_1 引脚上的设备的局部总线请求的优先级别要比 8087 本身所产生的请求的优先

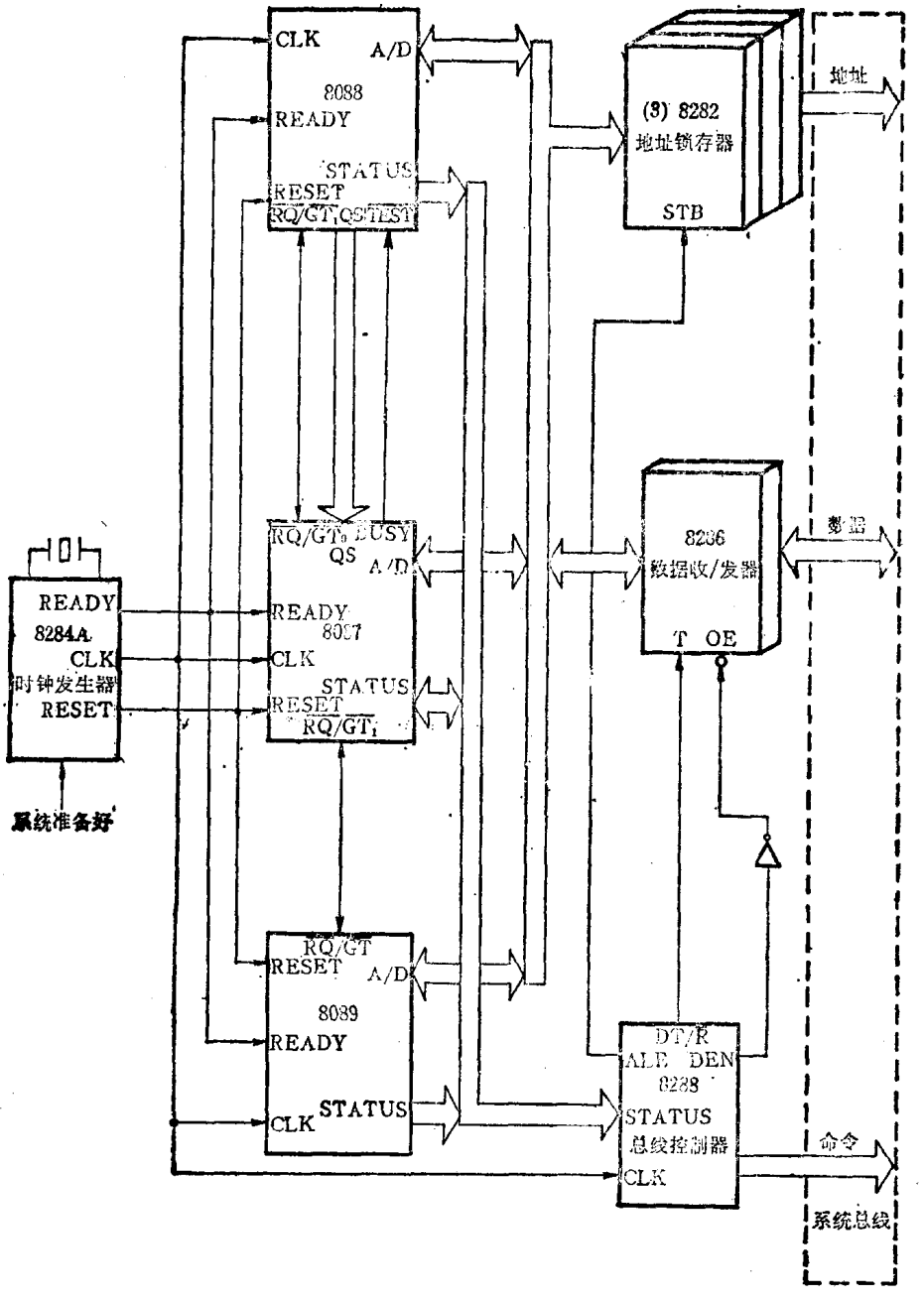


图 10.21 iAPX 88/21 结构框图

级别为高。如果当时 8087 掌握着总线控制权，而 8089 在这种情况下发出了一个局部总线请求信号到 RQ/GT_1 上，那么，8087 就在完成其当前的存贮器周期之后，把总线控制权让给请求者(8089)；若请求信号到达时，8087 不是总线控制权的所有者，那它就通过 $RQ-GT_0$ 引脚将请求信号传送给主机。

对主机为 8086 或 8088 的系统，表 10.19 列出了接在 8087 RQ/GT_1 引脚上的设备为了获得局部总线的控制权而需要的最大等待时间，表中给出的数值与主机何时把局部总线让给 8087 没有任何关系。

表 10.19 局部总线请求的最大等待时间(单位：时钟)

系统结构	没加锁的指令	加锁的交换指令	其它加锁指令
iAPX 86/21 (偶定位字)	15	35	max(15,*)
iAPX 86/21 (奇定位字)	15	43	max(43,*)
iAPX 88/21	15	43	max(43,*)

注：*代表最长的加锁指令的执行时间

表中列出的时钟数没有考虑总线周期中所插入的等待状态，所以，最大等待时间实际上还可能更长一些。从上表中可以看出，有三个因素决定了主机何时将释放局部总线：

(1) 主机的类型，8086 还是 8088？

(2) 当前正在执行的指令是什么？

(3) 是否使用了 LOCK 前缀？

3．iAPX 86/22、88/22

iAPX 86/22(或 88/22)在原 86/21(或 88/21)的基础上又增加了一个 8089，其 RQ/GT 的连接方式也与 86/21(或 88/21)相似，只是这时要有一个 8089 的 RQ/GT 与主机的 RQ/GT_0 相连。如图 10.22 所示。

在 iAPX 86/22 或 88/22 系统当中，两个 8089 对 I/O 请求的最大延迟时间是不相同的。在设计系统时，设计者必须考虑到各个 I/O 设备从请求 I/O 处理器 8089 为之服务，到 8089 能够响应该设备的 I/O 请求之间所允许的最大等待时间，从而确定哪个 8089 应为哪些输入输出设备服务，即确定把哪些设备接到 IOPA 上，又应把哪些 I/O 设备接到 IOPB 上。这是因为两个输入输出处理器的最大服务延迟时间之间的差距可能很大，至于使用主机的哪个 RQ/GT 引脚，其影响到并不太大。

不同的等待时间是由主机的两个 RQ/GT 引脚的总线响应之间的不能“抢占”特性所决定的，即因为在 IOPA 与 8087/IOPB 组之间不能进行“需要占用总线”的通信。IOPA 请求局部总线时的最坏情况是：等待主机、8087 和 IOPB 完成它们各自最长的存贮器周期序列；而 IOPB 请求局部总线的最坏情况为：等待主机/8087 和 IOPA 完成它们最长的存贮器周期序列，8087 对 IOPB 的最大等待时间的影响是很小的。然而，应该看到 8087 对 IOPA 延迟的影响却相当可观，特别是，当 8087 执行 FSAVE、FNSAVE 或 FRSTOR 这三条指令时，它需要的连续存贮器周期数为 50(主机是8086)或96(主机是8088)，可见，由于 8087 的存在，IOPA 的最大延迟时间可能变得相当长，以至于一般的应用都无法接受。但在 IOPB 看来，8087 的所有指令(包括上面

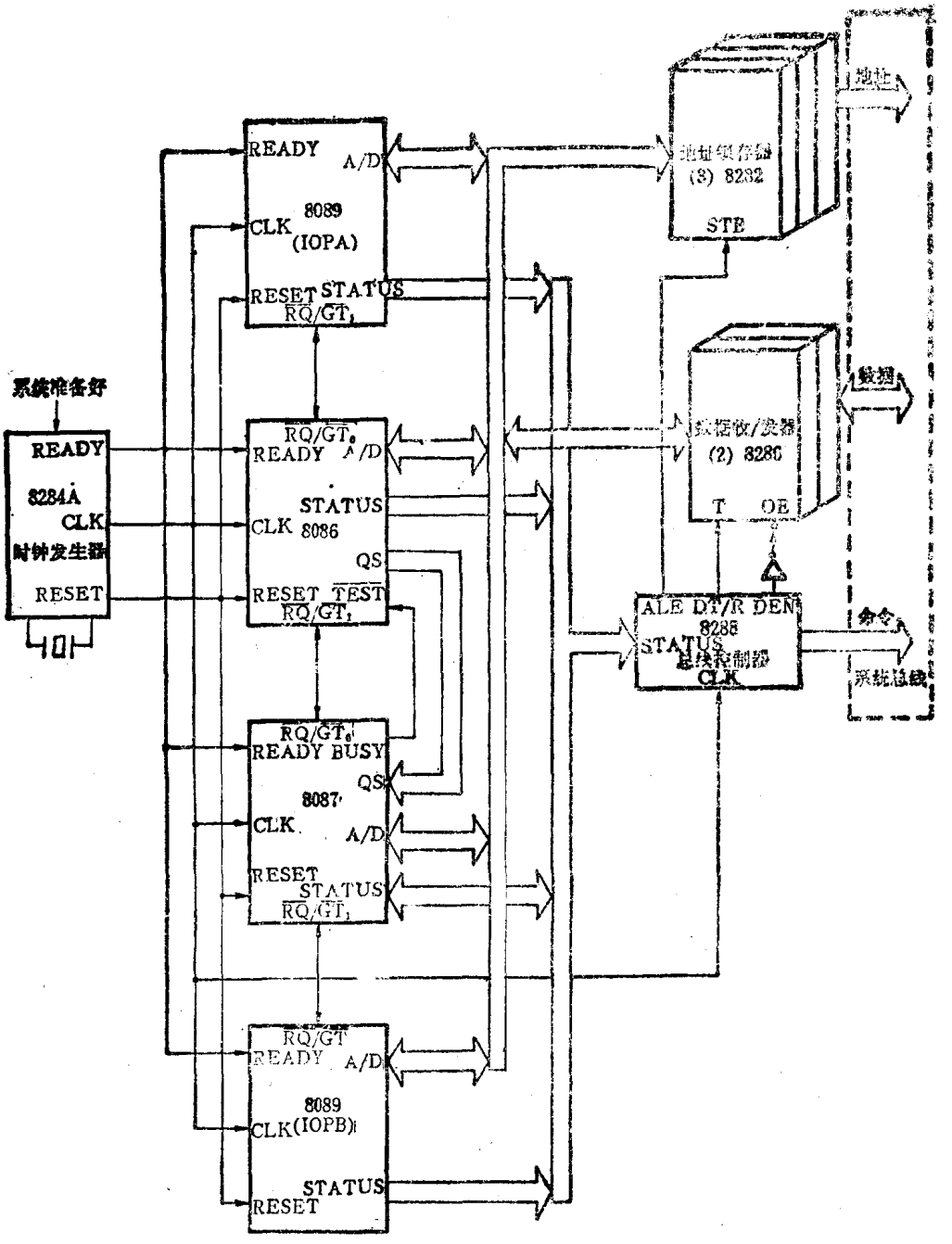


图 10.22 iAPX 86/22 系统结构图

提到的三条)都不会影响对局部总线请求的响应时间。8087 对 IOPA 产生这么大影响的原因就在于,8087 无法知道 IOPA 想要使用局部总线,因此,它也就不可能把总线控制权让给 IOPA,对于 IOPB 来讲,情况就不同了。对那些希望减小 IOPA 的最大等待时间的 iAPX 86/22、88/22 而言,也可以采取一些补救措施,即通过限制 8087 使用上述三条指令,从而达到减小 IOPA 最大等待时间之目的,因为 8087 完成其它指令所需要的时间不会超过 10 个(主机为 8086)或 16 个(主机为 8088)连续的存贮器周期,这就使得 IOPA 的最大延迟时间可以被许多速度要求不太高的应用所接受。至于 FSAVE、FNSAVE 和 FRSTOR 这三条指令的功能,则可以用其它的 8087 指令来模拟实现,由此产生的后果就是使得等价的保护(SAVE)和恢复(RESTORE)状

态操作的执行时间长了 427(8086 主机)或 747(8088 主机)个时钟周期。尽管如此，从整个系统的角度来看，这种方法还是有它一定的实用意义的。

但是应该注意，如果连接在主机的 RQ/GT_0 上的 8089 和连在 RQ/GT_1 上的 8087 同时申请总线，那么，总线将由 8089 所获得，也就是说，RQ/GT_0 的优先级别比 RQ/GT_1 要高，但是如果 8087/IOPB 已经占有了总线，那么 IOPA 就不能抢占总线了。

在 86/22 或 88/22 系统当中，确定哪个 I/O 设备连到哪个 8089 上的问题主要依赖于8089 对发出 I/O 请求的设备的响应速度，该设备所容许的最大等待时间必须大于 8089 的最大延迟与 8089 获得局部总线控制权的最大等待时间之和。假如任一个 8089 都不能提供足够快的响应时间的话，就要考虑采用 8089 的远程工作方式了。

以上讨论的是数值处理机的第一个结构问题，即 RQ/GT 的连接问题。当把 8087 加到 8086 或 8088 系统中去时，所碰到的第二个问题就是要确定把 8087 的 INT 信号接到哪里去。8087 的中断引脚 INT 是由软件选择的数值错误的一个外部标志，这些错误一旦发生，数值操作程序就将停止执行，直到对所出现的错误进行过处理时为止。将 INT 信号送往何处这一抉择对系统中其它的中断处理程序可能产生重大影响。

8087 的所有中断申请都是由检测出的异常事件引起的。在 8087 当中，当某次操作企图使用一个无效操作数或产生一个无法表示的结果时，我们就说发生了某个数值错误；如果程序试图进行一次不正确的或有疑问的操作，那么 8087 总要指出这一事件。在 8087 中，诸如1/0、-1 的平方根、从某个空的寄存器中读数这样一些不合规定的操作，都将引起出错信息的产生。当某个数值错误发生时，存在着两种可能的解决办法，一种方法就是让 8087 本身对发生的错误进行处理，即执行所谓的片上缺省的规定动作，前一节中曾经给出过这些活动的结果；数值错误的另一种解决方法就是由主机来处理错误。由于每种数值错误都可单独地被屏蔽，所以，为了让 8087 去处理某个数值错误，只要在其控制字中把相应的事故屏蔽位置为"1"就可以了，即把该事故屏蔽掉，这样，即使这一事故发生时，8087 也不会发出中断请求信号。在某些数值错误被屏蔽时，8087 对所有这些错误都定义一个缺省的规定动作。例如，对精度事故来说，其缺省的规定动作是使用当时起作用的舍入规则，对结果进行舍入处理。为了安全起见，应该非常小心地选择由 8087 所完成的这些出错处理操作。

当某个异常事件发生时，只有在该事故没被屏蔽且 8087 允许发出中断时，才能发出中断请求，也就是说，仅当控制字中的中断允许屏蔽位及相应的事故屏蔽位均为 0 时，8087 才能发出中断请求。倘若该事故是被屏蔽了的，那么 8087 就按照屏蔽响应方式(见 §10.3)处理该事故，而不把状态字中的中断申请位置位。假如此异常事件没被屏蔽，但不允许 8087 发出中断(IEM = 1)的话，那么，8087 的动作就取决于主机当前是否处于等待状态(执行 WAIT 指令)，如果主机不是处于等待状态，则 8087 就假定主机当前不想被中断，而在主机想被中断时，它会允许 8087 发出中断的(利用 FNENI 指令清除掉 8087 的中断允许屏蔽位 IEM)，一旦允许 8087 发出中断请求，8087 就将使 INT 信号变为有效。若主机当时正处于等待状态之中，那么，就有发生死锁现象的可能，即存在着无穷等待的危险，为了避免出现这种情况，8087 将使 INT 信号变为有效，给主机一个中断信号，从而让其离开等待状态，即此时，8087 不受中断已被禁止(IEM = 1)这一限制。8087 的中断请求过程如图 10.23 所示。

任何一个数值指令，若它使用了某个无定义的操作数，则都将产生一个无定义的结果。从这个意义上讲，开始的无效操作的结果将延续至整个程序当中(只要后面用到这一结果)。与

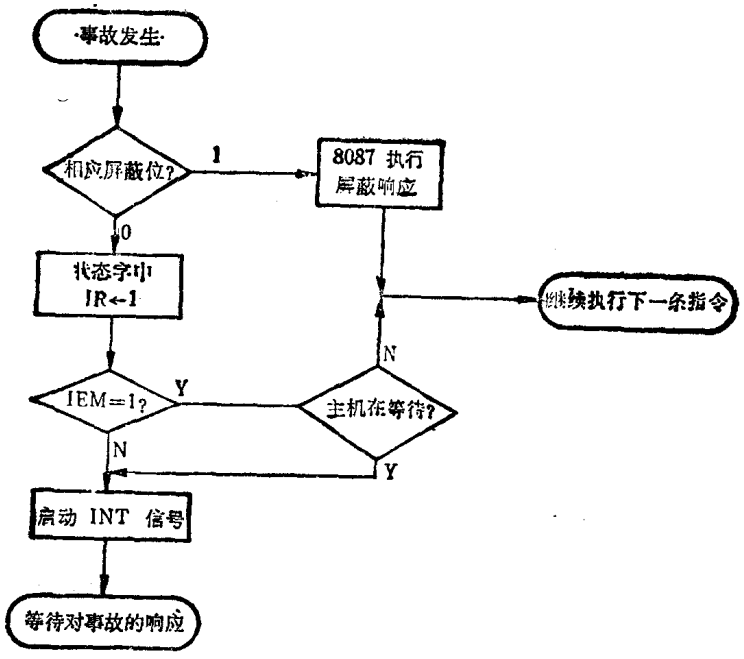
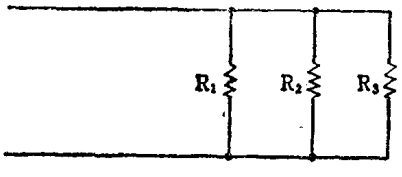


图 10.23 8087 的中断请求逻辑

此类似，将一个规格化数与一个非规格化数相乘所得到的结果亦为非规格化数。

主机软件对某种数值错误所需的响应要受到实际应用的影响，因此，在作出如何处理数值错误的决定之前，必须先了解实际应用的需要。在有些极为简单的应用当中，可以屏蔽掉所有的数值错误，这时，因为 8087 不会发出中断请求信号，所以，也就可以不连接 8087 的 INT 引脚。在这种应用当中，即使程序执行期间检测到某些数值错误，8087 仍然能够得出一个安全的结果，通过检测计算的最终结果，就足以判断计算过程是否有效。

对一些特殊的应用来说，程序不应该产生或使用任何无效操作数，更进一步讲，程序中用到的所有数值都应在规定的范围之内。若系统中的某个操作数或结果超出此范围，则指出发生了一次严重错误。当然，无效的输入数值、程序出错及硬件故障都将产生严重错误。在出现这种错误时，程序与硬件之间的一致性就难得到保障了，因此，需要立即对错误进行处理。此时，可以借助于 INT 信号来中断主机现行程序的运行，这种类型的中断应有较高的优先级。数值错误中断处理程序除了能对错误进行处理之外，还可以对系统的一致性进行检测，以使系统能够在一已知的安全状态恢复执行，处理程序一般不返回到出错点。未被屏蔽的数值错误对于测试程序来讲是非常有用的，在下一节的讨论中我们将可以看到，正确地利用同步手段，可以使程序员能够确定错误是由什么操作数、指令及存贮器值引起的。



等效电阻值 $R=\dfrac{1}{\dfrac{1}{R_1}+\dfrac{1}{R_2}+\dfrac{1}{R_3}}$

图 10.24 无穷大运算举例

一般来讲，除了无效操作数之外，我们可以屏蔽掉所有的其它数值错误，也就是说，8087 能够安全地处理上溢、下溢、被 0 除这样一些错误，这是因为在有些应用当中，无穷大的数值和被 0 除可能是允许出现的、是无害的。例如，在计算并联电阻的电阻值时，就会出现类似的情况，我们利用标准的计算公式(见图 10.24)，如果 R1 的值为 0，那么电路的电阻值也就变为 0，在屏蔽掉被 0 除及精度错误的情况下，8087 将得出正确的结果。

但是，对于某些应用来讲，一个数值错误可能并不意味着发生了一个严重的事故。其异常事件可能是指出某种硬件资源已经用光了，要求软件对之进行适当的处理，以弥补硬件资源之不足。这些事故不会经常发生，它们是由特殊的主机软件来处理的。我们也可以把这些异常事件看作是 NDP 功能的扩充，例如，规格化了的运算、把寄存器堆栈扩展到存贮器当中等处理皆属于这种扩充。

我们知道，主机仅有两个中断输入端，一个为非屏蔽中断 (NMI)，一个是可屏蔽中断 (INTR)。一般不提倡将 8087 的 INT 接到主机的 NMI 上去，这有两个理由，第一，NMI 不能被屏蔽，它常用于代表电源故障之类的重要故障；第二，Intel 公司为 NDP 所提供的软件不能支持 NMI 中断。而主机的 INTR 却允许中断在 CPU 中被屏蔽，利用 8259A 中断控制器又可以分离多重中断，此外，INTR 还具有 INTEL 后援。

数值错误中断不同于一般的指令错误中断，8087 发出的这种数值错误中断有可能在这次错误操作的 ESC 指令开始执行之后发生，而且延迟时间还比较长。例如，在主机启动一条数值乘法指令之后，它可能响应某个外部中断，并转去执行该中断的服务程序，而就在主机执行此中断服务程序时，若 8087 检测到一个溢出错误，在这种情况下，8087 的数值中断就成为一个先前的无关程序产生的结果了。因此，对 8087 中断的处理还有一些特殊的地方需要考虑，这将在后面讨论。

大多数系统中的软件都可被看作是由中断处理程序及应用程序两个部分组成的。中断处理程序用于对一些外部事件进行处理，当然，有时也用来响应 CPU 的内部事件。硬件中断控制器 8259A 负责检测外部事件，并通过给 CPU 一个 INTR 信号来调用相应的中断处理程序，即给 CPU 一个代码(中断码)，让其确定为该事件服务的中断处理程序。由于 8259A 一般管理着几种外部事件，故当几个外部事件同时发生时，就要借助于优先权分解技术，从中选出一个事件来通知 CPU，一般来讲，优先权较高的中断将先于低优先级的中断得到处理，甚至在优先级较低的中断处理程序正在执行时，若发生了某个优先级较高的中断，那么，原来的中断处理程序也将被中断，而转去处理优先级别较高的中断。

在多数系统当中，应用程序只有在没有外部事件需要处理时，即只有不在执行中断处理程序时，才能得到执行。这也就意味着，任何中断处理程序都比应用程序的优先级别高，因此，在识别出某个中断时，中断处理程序将取代原运行着的应用程序。但是，中断处理程序相互间的关系就比较复杂了。故在把 8087 加入系统当中时，其中断请求信号 INT 的连接方式可能会对系统中原有的中断处理程序产生影响，下面主要讨论有关这方面的问题。

假如我们希望在某段程序的运行过程中不要出现任何数值中断，那么，我们就可在这段程序的头上适当地设置 8087 的控制字，屏蔽掉 8087 的所有数值事故，这样，8087 就不会发出中断请求，当然也就不会影响这段程序的运行了。但要注意在这段程序的末尾，加上允许数值中断发生的指令。

从一般的意义上讲，中断系统具有快速响应外部事件、周期性地执行系统子程序这样一些特殊功能。而增加一个 8087 中断不应该影响中断系统的这一特色。8087 中断结构的理想目标可归纳为下面五点：

1．对于那些没有使用 8087 的中断处理程序来讲，不要让它们知道有关数值中断的发生情况。理由很简单，既然这些中断处理程序没有用过 8087，故它们也就不会引起数值中断，8087 也就没有理由中断它们。

2．避免给那些没有使用 8087 的中断处理程序增加代码，这些代码是专门用来防止 8087 发出中断请求的。这一问题只有对那些优先级比 8087 低的中断源才会发生，因为这些中断源对应的中断处理程序需要防止被 8087 的中断所取代。

3．在执行一个数值事故处理程序时，允许转去为其它优先级更高的中断服务。

4．对于那些使用 8087 的中断服务程序而言，应该为它们提供数值事故处理手段。

5．避免死锁现象的发生。

根据以上系统设计的目标，我们给出以下五种类型的中断结构，这样的五种结构已把8087的大多数应用情况包括进去了。

1．屏蔽掉 8087 的所有错误(事故)，使得数值中断不可能发生。此时，无需连接 8087 的 INT 引脚。

2．8087 是系统中唯一的中断源，即其它没有任何设备会向 CPU 发出中断请求，此时，直接将 8087 的 INT 引脚与主机的 INTR 输入端连接起来，而不需要使用 8259A 中断控制器，如图 10.25 所示。为了与 Intel 公司提供的软件兼容，总线驱动器给出一个中断向量 10H。

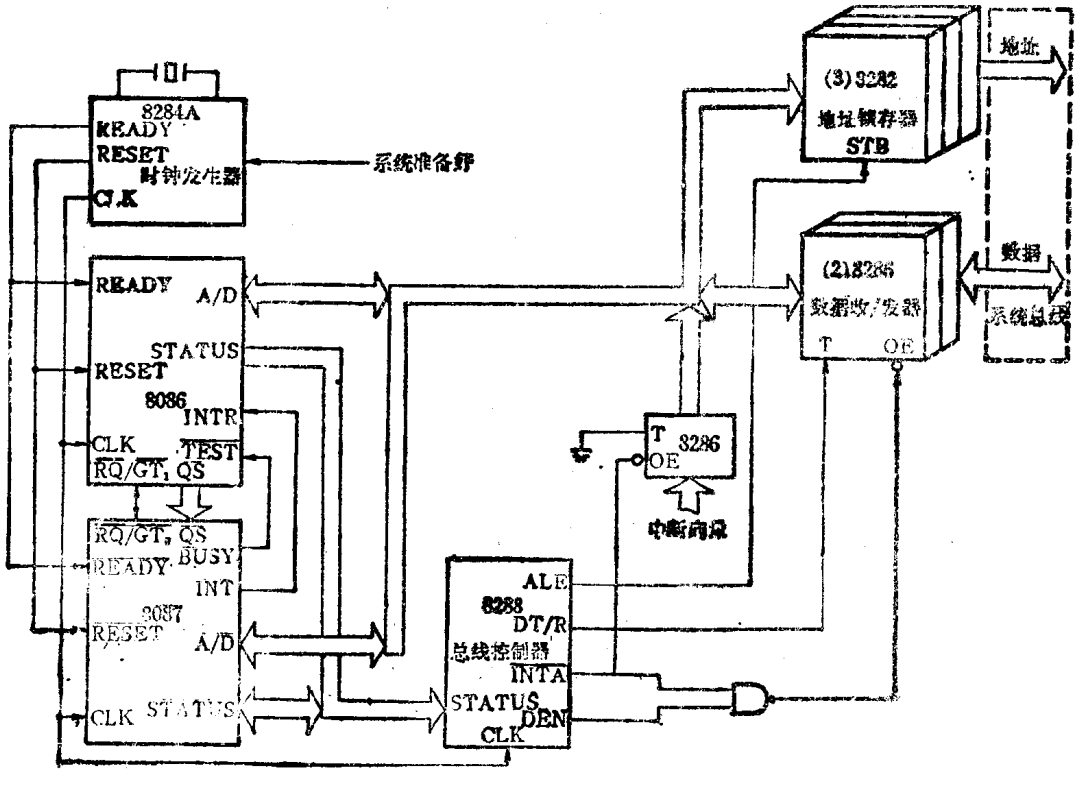


图 10.25　仅存在数值中断的 iAPX 86/20

3．8087 的中断将使其所有的活动停止，这时就得选择一个高优先级的中断输入，以终止一切有关的数值活动。这是一种特殊情况，因为中断处理程序不会返回到中断点，而是在清除系统后再重新启动，即不是继续执行原来的操作。

4．数值事故或数值程序错误有可能发生，但所有的中断处理程序要么不使用 8087，要么在屏蔽掉所有数值错误的条件下使用 8087。在这种中断系统中，数值错误使用优先级最低的中断输入，允许优先级更高的事件中断 8087 的中断处理程序，实际上，由于 8087 中断的优先级别最低，所以，8259A 中断控制器的优先级体制就能够在不另加代码的情况下，自动地起到

防止 8087 扰乱其它中断的作用。

S_1 除了中断处理程序本身可能产生数值中断之外，其它都与上一种结构方式相同。在这种系统当中，中断处理程序可以使用 8087，此时，8087的INT信号就应接到多个中断输入端，其中之一仍为优先级别最低的中断输入端，对于那些可能会产生数值中断的中断服务程序来讲，可以这样来连接 8087 的INT信号，即屏蔽掉优先级较高的数值中断，而允许低优先级的数值中断，这就相当于提高了该中断服务程序的优先级别。仅当中断处理程序在为一个需要进行 8087 事故处理的中断服务时，才不屏蔽优先级更高的中断输入。

采用以上这五种中断系统结构方式，就能够使得那些没使用 8087 的中断服务程序完全与 8087 隔离，而只有那些使用了 8087 的中断处理程序才需要完成一些与 8087 有关的中断控制活动。在多数系统当中，为了简化系统的设计，通常总是把比所有中断源设备都低的优先权分配给 8087，因为这样一来，就能保证 8087 的事故处理程序的优先级别比应用程序为高，同时也可以防止 8087 的事故处理程序与其它中断服务程序发生冲突。

§10.5 软件基础及程序设计技术

8087 作为 8086 系列一个协作处理器，它总是与主机处于多重处理环境之中的，即它们可以同时并行地执行各自的任务。与普通系统类似，NDP 数值软件系统可分成系统软件和应用软件两个大类，不管这些软件属于哪一类，也不管它们是通用的还是专用的，它们都要用到并发性、维持主机与 8087 的同步以及建立程序设计协议这样一些技巧。

§10.5.1 并 发 性

并发性是数值处理机的一个特色，它使得 8087 能够与主机一道同时执行各自的指令，其优点就是提高了程序的执行速度及系统的吞吐能力。数值处理机中所有的 Intel 高级语言都自动地提供了并发性能，并且能够自动地对并发过程进行管理；但对于汇编语言的程序员来讲，他就必须理解并发过程，并要对并发过程进行管理，还必须遵守后面将要讨论的同步规则。

在数值处理机中，并发性之所以可能得以实现，是因为主机与 8087 都具有它们自己的运算部件和控制部件，它们自己就能够确定某条指令该由谁来完成。应由 8087 完成的数值指令也放在主机的指令流中，其执行顺序也与主机取进的指令顺序相同。由于 8087 所完成的数值操作通常要比主机的操作花费更多的时间，因此，在 8087 完成一次数值运算的过程中，主机有可能已经执行了几条主机指令。

主机与 8087 之间的并发执行也是容易建立与维护的。我们可以把数值程序的活动分成程序控制和算术运算这样两个部分，程序控制部分所要完成的操作包括：确定要干什么、循环控制以及计算数值操作数的地址；算术运算部分所要执行的是加、减、乘以及对数值操作数的其它一些运算。主机和 8087 可以分别有效地完成这两个部分的工作。

因为在开始一个新的数值操作之前，运算和控制这两个部分必须集中到一个“良定”状态 (well-defined state)，所以，有必要对并发过程进行管理。这里的所谓“良定”状态是指：在此之前的所有运算和控制操作都已经完成，并且这些操作都是有效的。

§10.5.2 同步控制

在开始新的数值操作之前，主机通常要等待 8087 完成当时的数值操作，这一等待过程就叫做同步。对 8087 与主机的并发运行所进行的管理包括三种类型的同步：指令同步、数据同步和错误同步。其中指令同步和错误同步是由编译程序或汇编程序提供的，而数据同步则必须由汇编语言程序员或编译程序提供。

1．指令同步

由于 8087 每次只能完成一种数值操作，因而需要为指令同步提供保障。即在启动任何数值操作之前，必须保证 8087 已经完成了前面指令的所有活动。

主机的 WAIT 指令使主机能够在启动另一条数值指令之前，等待 8087 完成所有的数值操作。前面曾经讲过，每当 8087 在执行一条指令的时候，它就使其 BUSY 信号成为有效，这个信号被连到主机的 TEST 输入端（见图 10.18），当主机执行 WAIT 指令时，其动作为“当 TEST 有效时，就“等待”，这一指令对 8087 不产生影响。主机每五个时钟周期检查一次 $\overline{\text{TEST}}$ 上的信号，若 $\overline{\text{TEST}}$ 无效，主机就执行跟在 WAIT 后面的指令；若 $\overline{\text{TEST}}$ 有效，那就继续上述的等待——测试过程。这样，指令 WAIT 的执行时间就能从三个时钟周期(不需等待)延长到无限大，即 $\overline{\text{TEST}}$ 保持有效的时间有多长，指令 WAIT 的执行时间就有多长。由此可见，当 8087“忙”于处理其现行操作时，使用指令 WAIT 就能够阻止主机对下一条指令进行译码。

为了获得指令同步，在每条影响 NEU 的数值指令之前，都应该加上一条 WAIT 指令。以保证 8087 的 NEU 在接收下一条数值指令之前已经做好准备。除了 FNINIT、FNDISI、FNENI、FNSTCW、FNSTSW、FNCLEX、FNSTENV 和 FNSAVE 这八条处理器控制指令之外，其它所有的 8087 指令都将对 NEU 产生影响，为了简化 8087 的程序设计，8086 系列的语言翻译程序能够自动地在这些 8087 指令之前，加上 WAIT 指令。如果程序员用汇编语言写出这样两个语句：

```
FMUL
FDIV
```

那么，经过汇编程序之后，就将产生四条机器指令，相当于程序员写出：

```
WAIT
FMUL
WAIT
FDIV
```

这就可以保证乘法操作是在对除法指令译码之前完成。汇编程序的这一功能是通过在对8087指令的宏代码定义中加上 WAIT 指令实现的，前面，我们曾经给出过几个宏指令代码定义的例子。在这样定义过之后，汇编程序就能够自动地实现指令同步，当然，由于每条 WAIT 指令都需要一个字节的存贮器空间及 2.5 个时钟的平均运行时间，所以对系统发挥最大效能有一定的影响。

由汇编程序或编译程序提供的这种指令同步使得 NDP 能够并发运行。在非并发执行的程序当中，WAIT 指令是紧跟在数值指令后面的，这就使得主机在 8087 完成本次数值操作之前不能执行下去，即主机与 8087 不能同时执行不同的指令，显然，这样一来会延长整个程序的

运行时间．下面这个例子对并发和非并发程序的NDP运行时间进行了比较，对这样一段程序，采用并发技术时的运行时间是主机执行第二到第五句的时间与8087执行第一条和第五条语句所花时间中的大者，因为在8087执行第一个语句时，主机可以执行下面的语句；但是，采用非并发运行方式时，程序的执行时间是语句一到五的运行时间之和，因为这时主机和8087实际上不能并行工作，当8087还没有完成第一条乘法指令时，主机不能执行后面的MOV指令(第二条)．例子是这样的：

语句序号	并发执行方式	非并发执行方式
1	FMUL st(0),st(1)	NCMUL st(0),st(1)
2	MOV ax,size A	MOV ax, size A
3	MUL index	MUL index
4	MOV bx,ax	MOV bx,ax
5	FMUL A[bx]	NCMUL A[bx]

2．数据同步

数值协处理器8087能够分享主机的存贮器空间，这是前面已经介绍过的．但是，也应该看到，由于8087与主机之间的并发工作特性，使得它们两者有可能需要同时访问某个存贮器单元，有时会引起访问的正确性问题．图10.26给出了主机和8087共享某个存贮器单元I的四种可能情况．

```
情况1:                    情况3:
  MOV   I,1                 FILD   I
  FILD  I                   FWAIT
                            MOV    I,5
情况2:                    情况4:
  MOV   AX, I               FISTP  I
  FISTP I                   FWAIT
                            MOV    AX, I
```

图10.26 共享数据示例

图10.26中示出的前两种情况不需进行任何数据同步的特殊处理，因为在这两种情况中，在第一条主机指令完成其操作之前，它不可能启动下一条数值指令，即在数值指令开始执行时，前面的指令访问存贮器的工作已经结束．但在后面两种情况下，就都需要一条FWAIT指令来实现数据同步功能了，即主机在使用某个存贮器操作数之前，必须等待8087完成该存贮器操作数的准备、处理工作．情况3中的FWAIT指令迫使主机在改变I的值之前，等待8087从I单元中取数操作的完成；在情况4中，FWAIT指令防止主机在8087把I单元的内容设置好之前从I单元中取数．

很显然，程序员必须能够识别哪种形式需要明确的数据同步，哪种形式不需要这种数据同步．在一条数值指令的运行期间，若主机和8087使用的是不同的存贮器操作数，则无需考虑数据同步问题．倘若程序员能在汇编语言这一级正确识别以上四种情况的话，他就能够提高系统运行的并发性程度，且能保证操作有效．当然，由程序员来识别何时需要数据同步这种方法也就对程序员提出了比较高的要求，对相当一部分程序员来讲，这一要求可能是太高了，一时确实难以掌握这一方法，因而其应用也就受到了一定的限制．值得庆幸的是，还有两种能够自动地实现数据同步的方法，一是使用高级语言，由高级语言自动地建立并发操作，并对并

发过程进行管理，即在编译程序中解决数据同步问题。另一种方法就是以牺牲一些性能为代价，由汇编程序自动地实现数据同步，所采用的思想是很简单的，即让汇编程序在每条换码指令(ESC)的后面加上一个WAIT指令，当然，这时也就无并发运行可言了，实际上，执行的是非并发的程序。

用于实现数据同步的WAIT指令还能够用一个后继的数值指令来代替，因为在这条数值指令开始运行时，前面的数值指令肯定已经完成，其所有访问存贮器的操作自然也都已告结束，所以，所有的存贮器就都成为可用的了。但是，这时要注意的是，在以后某个时候可能要对程序进行修改，当然，也就有可能删去具有数据同步作用的数值指令，从而引起错误。例如，删去下例中的FMUL就会产生这样的一种错误：

```
FISTP   I
FMUL
MOV     AX,I
```

但是，指令FSTSW/FNSTSW、FSTCW/FNSTCW、FLDCW、FRSTOR和FLDENV的数据同步是一个例外，在8087执行这几条指令时，主机无需等待，而由8087自动地实现数据同步。实际上，当这些指令在运行时，主机不可能取得局部总线的控制权，当然也就不能改变存贮器中的内容，即解决了数据的同步问题。这些都是由协处理器接口所保证的，主机一旦执行到这几条指令，8087就立即向主机请求局部总线，在取得总线之后，8087要在它完成了现行的数值指令之后，才把总线还给主机。

从上面的讨论中可以看出，数据同步解决的问题是：在8087执行访问某个存贮器操作数的指令时，那么在8087完成该访问操作之前，保证主机不要访问这个存器贮单元。而指令同步解决的问题却是：在8087的NEU完成前一条指令的操作之前，一定不要开始下一条数值指令的执行。下面，我们用一个例子来说明指令同步及数据同步的实现方法。假设在执行这段程序之前，8087的寄存器堆栈中已经装进了操作数，当时，8087的NEU处于空闲状态，变量ALPHA及BETA的类型皆为整数字。例子是这样的：

```
            FMUL
            FSQRT
            CMP     ALPHA,100
            JG      CONTINUE
            MOV     ALPHA,100
CONTINUE:   FIST    BETA
            FWAIT
            MOV     AX,BETA
```

对应于上例的指令执行序列如图10.27所示：

例中最前面的两条指令是8087指令，ASM-86汇编程序将在它们的前面各加上一个WAIT指令，如图10.27所示，在主机遇到第一个WAIT指令时，由于8087当时不忙，所以CPU就与8087一道立即对下一条指令(乘法指令ESC-FMUL)进行译码，8087开始做乘法并发出BUSY信号，而CPU在执行完ESC指令之后，就执行第二个WAIT指令，因为这时连到8087上的TEST是有效的，故CPU就继续执行等待指令，直到8087执行完乘法运算，把BUSY信号降下来为止。然后，CPU和8087又译码开方指令(ESC-FSQRT)，8087开始求平方根，由于CPU

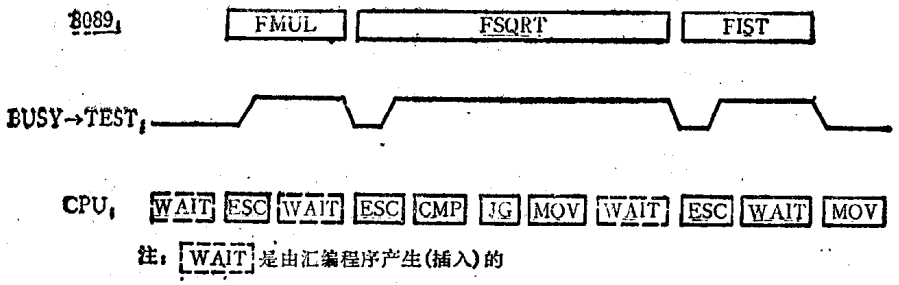


图 10.27 同步执行举例

完成 ESC 远在 8087 完成 FSQRT 之前，在这期间 CPU 不是等待，而是执行 CMP、JG、MOV 三条指令。然后，CPU 又遇到了汇编程序为指令 FIST 所加上的 WAIT 指令，等待 8087 开方操作 (FSQRT) 的完成，在 8087 完成 FSQRT 之后，CPU 又与 8087 一道执行下一条指令(ESC-FIST)，这又是一条 8087 指令，CPU 在执行完 ESC 指令之后，就开始执行程序员给出的 FWAIT 指令，这个 FWAIT 指令起数据同步的作用，它可以保证在 8087 已经把单元 BETA 的内容设置好了之后，CPU 才能从 BETA 中取数，并送至寄存器 AX 当中。

3．错误同步

几乎所有数值指令在运行期间的任一时刻都有可能引起数值错误。在 §10.4.3 当中，我们曾经讲：根据实际应用的需要，同一个数值错误可以有不同的解释，其处理方式也可以不一样。但是，下面的讨论适用于 8087 的所有用户，它不依赖于具体的应用，讨论的问题有：为什么需要错误同步？和如何获得错误同步？

象前面介绍的指令同步和数据同步一样，8087 与主机的并发运行也要求错误同步。实际上，数据同步与指令同步已经自动地提供了错误的同步机构．所谓错误同步是指：在发生某个数值错误的时候，能够保证在此之前的所有主机操作及 8087 操作都已经完成，且都是有效的。

如果屏蔽掉了所有的数值错误，那么，8087 就为这些数值错误执行各自的屏蔽响应，即执行相应的缺省的规定操作，这种规定操作是被简单地当作引起错误的数值指令的一个部分来处理的，即在错误发生时，由 8087 本身对之进行适当的处理，这就相当于延长了这条数值指令的执行时间，而对外界不产生任何影响，即采用这种错误处理方法时，8087 即使检测到某个数值错误，它也不给出任何外部标志，而只是在 8087 状态字的对应位上做一个标志，表示这个错误曾经发生．显然，假如 8087 对所有的错误都这样进行处理，那么也就不存在错误同步的必要了，但这并不是忽略错误同步的理由。

当某个未被屏蔽的数值错误发生时，8087 就将停止数值指令的进一步运行，且在 INT引脚上发出一个信号表示该事件。8087 的 INT (中断请求)信号通常都是送到主机的中断系统，实际上，中断主机的活动也就是 8087 请求主机帮助的一个过程，即要求主机对所发生的数值错误进行处理。而错误同步的作用就是：保证在某个未被屏蔽的数值错误发生之后，NDP 处于"良定"状态，即在此之前的所有主机和 8087 操作都已告完成且均为有效．只有这样，才可能确定错误为何发生、在何处发生这样一些问题，也才能够使程序正确地恢复执行。可以这样讲，错误同步的核心就是为正确地分析错误提供保障。

如果 NDP 允许并发运行的话，那么，在主机识别出数值中断时，主机的状态对该中断的处理是没有什么意义的，这是因为在主机被中断时，它可能已经破坏了执行数值指令(引起数值

中断的指令)时的现场,即改变了本身的许多内部寄存器的内容,甚至有可能在执行一个完全不相干的程序,例如正在执行某个别的外部中断处理程序。因此,利用主机的状态分析不了数值错误发生的原因,甚至不能确定该错误是由哪条数值指令所引起的。为了实现数值错误的分析与处理,8087专门设置了几个特殊寄存器,即所谓的事故指针,这些寄存器中存放的内容有:数值指令的地址、操作码以及操作数地址,在每开始一条数值指令的执行时,都将修改这些寄存器中的内容。可见,利用这些寄存器就可以来描述数值程序的状态,特别是描述执行失效指令时的状态,即当出现数值错误时,可以通过这些寄存器找到引起这一错误的指令或指令操作数,进而实现对这些错误的处理。

除了程序员的努力之外,"良定"状态对于错误恢复子程序来讲也是极为重要的。错误恢复子程序能够改变 8087 的运算规则和程序设计规则,而这些改变又可以重新定义 8087 对错误所作出的屏蔽响应、改变 8087 对程序员的表现形式、或改变 8087 上运算的定义方法。例如,可以把非规格化事故的响应规定为:自动地规格化从存贮器中取出的所有非规格化操作数;而把寄存器堆栈溢出的响应规定为:把寄存器堆栈扩展到存贮器当中,即提供由"无限个"数值寄存器所组成的堆栈,这就改变了8087对程序员的表现形式;此外,对于精度或溢出事故,也能够使 8087 的运算自动地提高变量的精度或扩大变量的取值范围。所有这些功能都是由 8087 通过数值错误及其有关的恢复程序,以对程序员透明的方式提供的。但应该注意,倘若没有正确的错误同步,数值程序就不一定能够正确地执行。例如,下面这段程序:

```
FILD    COUNT
INC     COUNT
FSQRT
```

这三条指令所做的工作就是从 COUNT 中取出一个整数、计算其平方根、并将 COUNT 中的整数值加 1。如果在执行 FILD 指令期间没有发生错误,那么,8087 的协处理器接口就能够保证这段程序的正确运行,从而得到所需要的结果。但是,如果在执行指令 FILD 时,寄存器堆栈发生溢出,情况就不同了,为了把 8087 的寄存器堆栈扩展到存贮器当中,8087 就指出相应的未被屏蔽的事故已经发生,该错误的恢复子程序必须识别这种情况,扩充寄存器堆栈,然后恢复执行开始的操作,但随之而来的问题就在于:不能确定 COUNT 的内容在 8087 中断主机之前是否已经加过 1 了,假如说在中断主机之前,COUNT的值已经加过 1,那么,在中断处理程序结束后,将重新执行 FILD 指令,而这时所取出的单元 COUNT 的内容就比原来应取的值大 1,从而导致程序结果的错误。之所以会出现这种情况,则是由于程序中没有实现错误同步,因为在 FILD 指令引起错误时,由主机所执行的 INC 指令就不再是有效的了,即不满足错误同步的条件。

错误同步的实现依赖于主机的 WAIT 指令和 8087 的 INT 及 BUSY 信号。当 8087 中出现一个未被屏蔽的错误时,8087 就将发出 BUSY 和 INT 信号,其中,INT 信号用于中断主机,而 BUSY 信号则主要用来防止主机破坏当时 8087 的环境,BUSY 信号在那条引起错误的数值指令执行期间总保持高电位,这样,实现指令同步功能的 WAIT 就能够防止主机在错误得到处理之前启动另一个数值指令。在对数值错误进行处理时,主机负有两个责任,其中之一就是在它检测到一个数值错误时,绝对不能破坏8087当时的环境,这一点可以通过起指令同步作用的 WAIT 实现;主机应负的第二个责任就是必须清除掉该数值错误,并使程序从该错误中恢复过来。由数值错误唤醒的恢复程序在对错误进行了适当的处理之后,可以恢复原来程序的

运行、给程序员显示 NDP 的当时状态、或让程序流产。在 INT 和 BUSY 信号均为高电平时，8087 不能完成任何有用的操作。

必须注意，当主机不能被 8087 中断，而 8087 又发出一个 INT 信号时，就可能发生死锁现象。具体讲，当主机正在执行 WAIT 指令而它又不能识别由 8087 产生的中断请求时，因为这时 8087 的 BUSY 信号使得主机不能执行后面的指令，从而造成 8087 等待主机为它的故障服务，主机等待 8087 完成当前的数值操作，两者在相互等待，谁也前进不了，进入无穷等待的死锁状态。除非有某个外部事件中断主机，否则将一直保持这种局面，如果发生了其它中断的话，NDP 就能够转去完成一些其它功能，但是其数值程序仍将保持在"僵死"状态；如果系统中不可能出现其它中断，那么 NDP 将永远等待下去。

从上面的分析中我们可以看出，在 8087 到主机的中断通路不通的情况下，死锁现象就有可能发生。发现中断通路上的断点也还是比较容易的，8087 的中断请求可能会在下面这三个地方被阻塞：主机中的中断允许位被屏蔽(IF = 0)，这时，主机拒绝接收连接于 INTR 引脚上的一切中断请求，显然，8087 也就不能中断主机；第二个阻塞点是 8259A 中断控制器，假如 8087 的 INT 是被 8259A 显式屏蔽了的中断请求，即将 8259A 的中断屏蔽寄存器(IMR) 的相应位置位的话，那么，8087 的中断请求也就不能被送到主机；此外，8259A的优先权分析机构也能够阻碍 8087 的中断请求，即由于主机正在为一个优先级别较高的中断服务，从而相当于屏蔽了 8087 的中断请求。当然，有些硬件故障也可能成为中断通路上的断点。

用户编制的事故处理程序本身也有引起死锁的可能。在事故处理程序的运行过程中，8087 的 BUSY 信号总是处于有效状态(为高电平)，所以 8087 在此期间不能执行其它动作，而是在等待事故状态的清除。在这种情况下，如果事故处理程序中出现了 WAIT 指令或以 WAIT 为前导的任何数值指令的话，就会导致无穷等待，即发生死锁现象。因此，在编制事故处理程序的时候，必须把这一因素考虑进去，要么不要使用指令 WAIT，要么在清除了 8087 的事故状态标志之后，再使用带有WAIT 的指令。一般说来，在 CPU 的中断被屏蔽而 8087 的 BUSY 信号可能为有效时，不应该执行带有 WAIT 前导的指令或指令 FWAIT.

就一般情况而言，应用程序员无需考虑死锁问题，这是因为一般的应用程序在其运行过程中不屏蔽数值错误，即所有的数值错误都能够中断程序的执行，在这种情况下，不可能发生死锁现象.但在系统软件或中断处理程序运行期间，数值中断可能被屏蔽了，此时就要注意防止死锁现象的发生了。这里应该遵守的一个重要规则就是："如果 8087 的中断通路可能不通，且可能出现某个未被屏蔽的数值错误的话，决不要执行等待 8087 (WAIT 指令)的操作".

综上所述，我们可以看出，NDP 当中的同步要求，实际上是为获得并发性而付出的代价。Intel 的高级语言本身提供了并发性及同步功能，所以，高级语言程序员无需考虑这些问题，而对汇编语言程序员来讲，则能够在使用并发性与不使用并发性两种方案中做选择，若紧接着每条数值指令之后加上一个 WAIT 指令的话，就能够避免对同步的要求，但也就牺牲了系统的并发功能。

§10.5.3 程序设计技术

8087 提供了一个面向堆栈的寄存器集，它也具有面向寄存器堆栈的数值操作指令。为了满足数值程序和通用程序的需要，NDP 的寄存器和指令集都经过优化处理，主机提供通用数

据处理所需要的指令和数据类型，而 8087 则提供专门用于数值运算的数据类型及指令。

由于数值程序与通用程序的需要不同，因而 8087 所识别的那些指令与数据类型也就不同于主机识别的指令与数据类型。数值程序中具有一些很长的算术表达式，其中，有些中间结果在几个语句中都要用到，也就是说，这些语句要多次引用某些临时值，为了提高运算的速度，一般应把这些中间结果放在 8087 的寄存器堆栈当中，而不是把它们送到存贮器当中去。8087 共有八个数值寄存器，这些寄存器的寻址都是相对于栈顶指针 ST 进行的，栈顶指针 ST 在 8087 的状态字中被定义，数值指令给出的寄存器地址要加上 ST 的值才形成内部的绝对地址，这种数值寄存器的相对寻址方式的优点类似于存贮器操作数的相对寻址，它使得程序在不了解寄存器的分配情况时，使用数值寄存器。我们也可以把这种寻址方式分为两种模式，一种模式就是把寄存器栈顶用作操作数，即隐含寻址方式，这时，数值指令当中不需要任何寻址位，它也就不具有寻址的灵活性，因而只有一些特殊指令才使用这种寻址寻址模式。另一种寄存器堆栈的寻址模式是这样的，它允许指令把栈中的任一其它寄存器与栈顶寄存器一起使用，其中之一用作目标操作数，大多数双操作数数值指令都采用这种模式。

当然，任何实际资源总是有限的，8087 的数值寄存器也只有八个，因此，在通常情况下，程序员就必须确保在任何时候压入数值寄存器堆栈的数不超过八个，但是，利用软件可以部分地解决硬件上的一些限制，例如，软件可以提供“虚拟”数值寄存器，将寄存器堆栈的规模扩展到 6000 个甚至更大。当寄存器堆栈上溢或下溢时，就产生一个数值错误，若该错误没被屏蔽，8087 就发出中断请求信号，对应的中断处理程序就根据实际需要，确定是该把数值送到存贮器，还是该从存贮器中读进数值，实际上，该中断处理程序解决的就是数值寄存器堆栈的存贮器映象的管理问题。

在对 8087 的寄存器堆栈有了一个较好的理解之后，接下来我们讨论几个有用的程序设计协议，下面这些协议能够保证与 Intel 的支持软件及高级语言兼容。

1．如果没把数值寄存器堆栈扩充到存贮器当中，那么，程序员就必须确保存放在 8087 寄存器堆栈中临时值的数目不超过八个。为了提供足够多的可用数值寄存器，可以把某些数值存放到存贮器当中。

2．在调用子程序(过程)时，可以利用数值寄存器堆栈，把头七个数值参数传送给子程序，而把其它参数送到主机的堆栈当中，其参数的传递次序是从左到右逐个进行的。8087 寄存器堆栈中的第八个寄存器用于数值运算。当子程序结束时，堆栈中的所有参数都将被弹出来。

3．返回的所有数值都将放在 8087 的寄存器堆栈当中。

4．在任何子程序结束之前，8087 都必须完成所有的存贮器读/写操作，这就为正确的数据同步和错误同步提供了保证。

5．8087 的工作模式是由其控制字所定义的。在调用子程序时，倘若子程序需要使用一个与调用者不同的工作模式，那么，该子程序就应该首先保护现行控制字，然后设置新的工作模式，并在子程序结束时，恢复原来的控制字。

§10.5.4 iAPX86/20、88/20 的程序设计

作为本章的结束，我们准备讨论几个程序设计的例子，通过这几个例子的讲解，来说明

NDP 的一些程序设计技术以及常用功能。给出的几个例子都已经通过了编码、汇编及调试这几个阶段，当然，这并不意味着它们的正确性已经得到保障，因为"调试只能找出程序中存在的错误，而不能证明程序的正确性"。

下面将要介绍的五个程序设计例子是这样的：保护、恢复 8087 的状态环境；在不使用指令 FSAVE/FNSAVE 的条件下，保护 8087 的状态环境；将浮点数转换成 ASCII 代码；将 ASCII 码化为浮点数；三角函数的计算。这些例子提供了使用数值处理机所需要的一些基本功能，无论是 8087，还是 8087 的模拟程序，都可以处理这几个程序，且不需要对源程序作任何修改。

操作系统以及那些可能用到 8087 的中断处理程序都要进行 NDP 的状态切换，因此，它们将要用到程序段例 1 或例 2；为了使输入输出数据便于阅读，就需要进行浮点数与 10 进制的 ASCII 代码之间的转换，即把机器内部的 2 进制浮点数结果转换为 10 进制 ASCII 码供输出显示(例 3)，而把以10进制 ASCII 码表示的输入数据转换为机器内部的 2 进制浮点数格式(例 4)；至于例 5，那是一个有关三角函数计算的例子，借助于它，可以帮助计算出正弦函数或余弦函数的值。

例 1：

本例实现数值环境(即 8087 状态)的切换，并且能够保证在其运行期间不出现死锁。我们知道，操作系统为了进行任务调度(进程调度)需要进行系统的状态切换，同样，用到 8087 的那些中断处理程序也需要实现 8087 状态之间的切换。这种状态切换总是由两个基本部分组成的，即数值环境(8087 状态)的保护和数值环境的恢复。同时，这些功能的完成不应该依赖于 8087 的当前状态。

在这个例子当中，我们给出了两种 8087 状态的保护方法，实际上，两种保护方法之间的主要区别就在于它们使用了不同的状态保护指令，一种方法借助于指令 FNSAVE，另一个则使用了指令 FSAVE。应该选用哪条指令则由主机中断的状态所决定。

1．使用 FVSAVE 来保护 8087 的状态

在主机中断被屏蔽了的时候，就应该使用FNSAVE 指令来保护8087的状态。因为这样一来，在开始保护状态之前，主机就不必等待 8087完成其现行操作，即无需提供指令同步(指令FNSAVE 不带前导指令WAIT)，从而就避免了任何潜在的死锁现象，不会发生无穷等待的事件。

8087 的总线接口单元(BIU)在遇到指令 FNSAVE 时，就将其保存起来，等待 8087 的算术运算部件(NEU)完成其当时的操作，一旦 NEU 变为空闲，BIU 就将启动 NEU，令其执行状态环境的保护操作。在 BIU 等待 NEU 完成其当前操作的同时，主机可以执行指令 FNSAVE 后面的其它一些非数值指令。在状态环境保护指令执行之后，存贮器中就将形成一个由 47 个字组成的区域，其中存放着 8087 当时的状态环境，存放格式在第 3 节有关指令系统的介绍中已经给出。总之，在主机中断被屏蔽的情况下，为了防止死锁的发生，我们用如下程序段来保护 8087 的状态环境：

```
no-int-NPX-save:
  FNSAVE  save-area;  Save NPX context. (保护 8087 的状态环境)
  FWAIT               ; Wait for save to finish.(等待状态保护操作
                        的完成)
```

程序中的 save—area 需要 47 个字 (94 个字节)的内存空间，指令 FWAIT 的作用是提供

系统的指令同步和数据同步．在执行 FNSAVE 指令的时候，必须屏蔽掉主机的中断，以避免重复调用 FNSAVE 指令，这是因为 8087 的 BIU 每次只能保存一条 FNSAVE 指令，如果说不屏蔽掉主机中断，那么中断就有可能发生，这样的一个主机中断也就可能需要执行第二个 FNSAVE 指令，从而破坏了保存在 8087 BIU 中的前一条 FNSAVE 指令．但是，我们并不提倡仅仅为了执行一条 FNSAVE 指令而屏蔽掉主机的中断，一般来说，这样做不是最好，有时甚至是不允许的．那么，是否存在较好的解决方案呢？答案是肯定的，也就是说，存在着一种 8087 状态环境的保护方法，在状态的保护期间，允许主机中断发生，只不过这时我们建议使用 FSAVE 指令，而不是使用 FNSAVE 指令．

2．利用 FSAVE 指令实现 8087 状态环境的保护

指令 FSAVE 所完成的操作与 FNSAVE 相同，但是，它利用了指令同步特性，即在主机启动状态保护操作之前，它必须等待 NEU 进入空闲状态．由于主机不理睬在完成一条 WAIT 指令与开始其后继的换码指令之间出现的任何中断，故一旦 NEU 降下 BUSY 信号，它就立即准备执行状态的保护操作，这样，在 BIU 中也就不可能出现多次保护状态的要求．但是，因为主机要执行 WA IT指令，从而必须考虑死锁的防止问题．为了在使用 FSAVE 指令时避免死锁，就要防止 8087 在出现未屏蔽错误时发出 BUSY 信号，应该看到，适当地设置 8087 的控制字，就可以达到上述目的，这是因为当控制字中的 IEM 字段的内容为 1 时，即使发出了某个未被屏蔽的错误，8087 也不会由此发出 BUSY 信号或 INT 信号．第三节中已经介绍过，8087 的 FNDISI 指令所起的作用就是设置控制字中的 IEM．利用 FNDISI、FSAVE 和其它一些指令就能够完成 8087 的状态环境保护操作，并能保证保护过程中不会发生死锁，它也不依赖于 NDP 的中断状态．

在指令 FNSAVE/FSAVE 执行之后，总需要获得标准的数据同步和指令同步，因此，我们总是在指令 FNSAVE/FSAVE 的后面加上一条 WAIT 指令．由于在执行状态保护之前，已经屏蔽掉了 8087 的所有错误，所以，FNSAVE/FSAVE 指令后面的等待指令是安全的，这是因为 8087 最终肯定要发出不"忙"信号(因为 8087 不会发出中断请求，等待主机为之服务)，故不可能发生死锁．

下面这段程序就使用指令 FSAVE 来保护 8087 的状态环境，它也是一个比较通用的 8087 状态保护程序．它的运行不受 NDP 中断状态的影响，不会引起死锁的发生．变元 save__area 需要 47 个字的内存空间．

```
NPX__save:
    FNSTCW      save__area (保护控制字中 IEM 字段的状态)
    NOP                    (延时，此时 8087 在保护控制字)
    FNDISI                 (禁止 8087 发出"忙"信号)
    MOV     ax,save__area  (获得原控制字)
    FSAVE       save__area (保护 8087 的状态环境)
    FWAIT                  (等待状态保护操作的完成)
    MOV     save__area,ax  (将原控制字送至保护区中)
```

3．利用 FRSTOR 来恢复 8087 的状态

8087 在执行指令 FRSTOR 时，能够自动地防止主机干涉其读存贮器的操作，因此，在用 FRSTOR 指令恢复 8087 的状态时，不需要考虑数据同步问题．又因为指令 FRSTOR 本身不会

引起错误，故错误同步也就不是必需的了，但是，应该注意，恢复过来的状态有可能指出某种错误的存在。倘若在 FRSTOR 之后，还要执行一些数值指令，并且不知道 8087 新状态字中的错误标志情况，那么在数值事故不能中断主机的情况下，就有发生死锁的可能。反之，即如果在允许8087 的数值错误中断之前，不执行任何别的数值指令，那么死锁就不会发生。假设变元sa_vearea 中保存着 8087 前面某个时候的状态，那么，利用下面这条汇编语句就可以把先前保存在save_area 中的 8087 状态恢复过来：

```
FRSTOR  save_area
```

例 2：

本例以另一种方法实现 8087 状态的切换，即在不使用指令 FSAVE/FNSAVE 和 FRSTOR 的条件下，实现 8087 的状态转换。正象§10.4.3 中所指出的那样，iAPX 86/22 或 88/22将从这一改变中获得益处，但是，这一改变也降低了数值环境的保护/恢复速度。采用这种方法之后，程序就不再是在一个很长的存贮器传送序列中保护或装入 8087 的整个状态信息，而是通过使用 FSTENV/FNSTENV/FLDENV 指令以及一些数值寄存器存/取指令，分步骤保护或恢复 8087的状态。

1．与 FSAVE/FNSAVE 的一致性

不管采用哪种方式，保存在存贮器中的 8087 状态格式必须是一致的，即其格式必须与指令 FSAVE/FNSAVE 产生的结果相同。为了利用 FSTP 指令保护数值寄存器的内容，由于在一般的情况下，8087的八个数值寄存器并非全被占用，所以在使用 FSTP指令之前，必须先把这些寄存器标志为有效、为 0 或特殊值，但不可以把任意一个数值寄存器标志为空，只有这样，才能使保护起来的状态格式符合要求。在本例当中，所有的寄存器都被标志为有效，而不管它们的内容及原始标志为何，然后，再逐个地把这些寄存器的内容送到状态保护区的适当位置上，这样，即使某些寄存器原来为空，FSTP 指令也还是把它们保护起来，从而，保证结果与指令 FSAVE/FNSAVE 一致。在将所有的数值寄存器都保护好了之后，就把它们的标志位都置为空，这也是与执行完 FSAVE/FNSAVE 之后的动作一致的。

下面这段程序完成 8087 状态的保护操作，其中没有用到 FSAVE 或 FNSAVE 指令，分析一下这段程序，就可以看出，它所形成的结果是与指令 FSAVE/FNSAVE 相同的。

```
Small_block_NPX_save:
    FNSTCW   save_area          (保护现行的 IEM 状态位)
    NOP                         (延时，8087 正在保护控制字)
    FNDISI                      (禁止 8087 发出"忙"信号，以防死锁)
    MOV      ax,save_area       (获得原控制字)
    MOV      cx,8               (设置数值寄存器计数器)
    XOR      bx,bx              (标志字，以将所有的数值寄存器都标志为有效)
    FSTENV   save_area          (保护 8087 的数值环境)
    FWAIT                       (等待指令 FSTENV 的结束)
    XCHG     save_area+4,bx     (bx⇐原标志字，新标志字⇐0，即所有寄存器都为有效)
    FLDENV   save_area          (获得新标志字)
    MOV      save_area,ax       (原控制字送到数值环境中)
    MOV      save_area+4,bx     (原标志字送到数值环境中)
```

```
    XOR     bx,bx                (设置数值寄存器索引的初值)
reg_store_loop:
    FSTP    saved_reg[bx]        (保护寄存器)
    ADD     bx,type saved_reg    (索引指针 bx 指向下一个寄存器)
    LOOP    reg_store_loop       (完成整个数值寄存器堆栈的保护)
```

2．与 FRSTOR 的一致性

8087 状态的恢复是与上面相反的一个过程。八个寄存器以与存入的相反次序读进，每装入一个寄存器，就将给该寄存器一个标志值；其实，这里设置的标志值没有什么作用，因为后面的装入环境指令 FLDENV 将重新装入标志字。

正确的状态恢复需要两个前提：所有的数值寄存器必须为“空”；ST 字段的内容必须与待恢复的状态中的 ST 字段相同。实际上，只有在该状态的保护与恢复之间所执行的压栈和出栈操作次数相同时，前述的两个条件才能得到满足。如果说上面的两个条件得到满足，那么就可以用下面这段程序实现 8087 状态的恢复(没有使用 FRSTOR)：

```
small_block_NPX_restore:
    MOV     cx,8                 (设置数值寄存器计数器的初值)
    MOV     bx,type saved—reg*7  (ST(7)的偏移量送至 bx)
reg_load_loop:
    FLD     saved_reg[bx]        (恢复寄存器)
    SUB     bx,type saved_reg    (指向下一个数值寄存器)
    LOOP  reg_load_loop
    FLDENV  save_area            (恢复 8087 的环境)
```

倘若前述的两个条件不能得到满足，那么，为了实现 8087 状态的恢复，就必须在上面这段程序前面，加上一段代码，以迫使 8087 的所有寄存器变为“空”(利用 FINIT 指令)，并且使状态字中的 ST 字段满足上面的第二个条件。下面就是应该加在上段程序前面的代码，其中，前面某个时候保存起来的 8087 状态放在 save_area 中，temp_env 是一个由 7 个字组成的暂存区域，用于构造 8087 的环境。

```
NPX_clean:
    FINIT                        (初始化 8087)
    MOV     ax,save_area+2       (取原来的状态字)
    AND     ax,3800H             (抽出 ST 字段)
    FSTENV  temp_env             (把 8087 的环境信息存放到临时区域 temp_env 中，其
                                  中，所有寄存器都标志为空，ST 字段的内容为 0)
    FWAIT                        (等待 8087 环境保护操作的完成)
    OR      temp_env+2,ax        (置上所需要的 ST 字段的内容)
    FLDENV  temp_env             (建立新的 8087 环境)
small_block_NPX_restore:         (开始 8087 状态的恢复操作)
    :
```

例 3：

在计算机的输入/输出处理当中，输入/输出字符通常是用 ASCII 代码表示的，一般来讲，

它们不同于机器的内部表示。那么，怎样才能把 8087 的浮点数结果以用户容易接收的形式输出呢？即存在着一个把浮点数转换成 10 进制的 ASCII 字符串的问题。下面这个程序实例给出了这一问题的一种解决方法，作为一个函数，它可以被 PL/M-86、PASCAL/86、FORTRAN/86 或 ASM/86 程序所调用。

整个转换程序由三个独立的模块组成。转换的绝大部分工作都是在模块 FLOATING_TO_ASCII 中完成的，之所以把转换程序分成三个模块，主要是考虑到另外的两个模块具有一定的通用性，即这两个模块可以由其它程序所调用，其中之一，即模块 GET_POWER_10，在将 ASCII 码转换为浮点数的子程序中也要用到；还有一个模块，即 TOS_STATUS，它起的作用就是确定数值寄存器栈顶的内容。

为了避免产生异常事件，程序当中也做了些考虑。本程序可以接收任何可能的数值，只有在数值寄存器堆栈的空间不够使用时，才有发生事故的可能。对于已经送到数值寄存器堆栈上的值，程序要检查其存在与否、类型(NAN 或无穷大)及状态(非规格化、未规格化、0 和符号)。字符串的长度具有上下界，对它们要分别测试。倘若寄存器栈顶为空，或者字符串过分短的话，该函数就将返回一个错误代码。在函数内部，对于太大或太小的数也进行了处理，以防止上溢或下溢。

通过这个例子，我们可以看到一些数值指令、8087 的数据类型及精度控制的使用情况·例如，自动地把浮点数化为 BCD 码的指令、计算 10 的乘幂、并发性的建立与维护、数据同步以及 8087 舍入方式的使用等等。

将 2 进制浮点数转换为 10 进制表示这一过程得依赖 8087 的 BCD 这种数据类型，至于将 BCD 的各位分解成一些单个 ASCII 码表示的 10 进制数并不困难，而转换过程的主要工作则在于把浮点数值换算成对应的BCD值。若要打印一个九位的结果，则需要把给定的数值换算成 10^8 到 10^9 之间的一个整数，例如，对于 +0.123456789 这样的一个数，需要 10^9 这个比例因子(换算因子)，才能产生可用九个 BCD 位表示的数值，即 +123456789.0。为了避免改变各位的数值，这种比例因子必须是 10 的整数次幂。

在本例当中，我们并不试图尽可能多地给出有效位数，而是希望形成比较短的字符串结果，并能达到一定的速度及精度要求。当然，假如再仔细一些，也能够获得更高的精度，不过程序也将更长些，其速度也有所降低。为了避免不必要的转换误差，程序总是力图保持整数在其表示范围之内。对于在给定的取值范围之内，且能用 10 进制数精确表示的所有数值，该程序也将得到精确的转换结果。假如给出的是一个整数，并且该整数在给定的结果字符串的表示范围之内，那么，就无需再对它取比例因子，而只要将它以 BCD 的形式直接存放起来。对于非整型数值而言，只要它能精确地用 10 进制数表示出来，并且其长度不超过字符串的长度之限制，那么，所得到的转换结果也是精确的，例如，对于 0.125 这个数，无论用 2 进制还是用 10 进制都可以精确地表示它，为了将该浮点数化为 10 进制数，只要取比例因子 1000，就可以获得结果 125。在对某个数值取比例因子的同时，本程序(函数)必须"记住"小数点在最终结果中的位置

从宏观上看，把一个 2 进制浮点数转换成 10 进制的 ASCII 字符串需要经过三大步，即确定数值的大小、取比例因子得到 BCD 数、将 BCD 数转换成 10 进制的 ASCII 字符串。第一步，即确定数值的大小，需要求的是数值 x，以使得该数值能用 $I*10^x$ 表示出来，其中 $1.0 <= I < 10.0$；第二步，即对原数值取适当的比例因子 10^s，使该数值与 10^s 相乘后得到一个整数结果，并使该整数所需要的 10 进制位数不超过 ASCII 字符串所提供的字符串长度；在取好了比例

因子之后，借助于8087的舍入方式及BCD转换指令，就能够把数值化为主机软件容易转换成ASCII字符串的形式。

为了解释清楚每个步骤，就必须注意一些细节问题。首先，并非所有浮点表示都具有数值意义，即转换程序有可能会碰到无穷大、无定义、非数值(NAN)这样一些值，因此，转换程序应该能够识别出这样一些特殊值，并对之分别进行处理。此外，还存在着一些特殊的数值，即未规格化数、非规格化数及伪0，它们虽然都具有一定的数值意义，但是，它们都表示在前面的某些计算过程中失去了一些精度，因此，转换程序也应该能够识别它们。

一旦确认某个浮点表示具有数值意义，并设置了适当的非规格化标志且使之规格化了之后，就必须对该数值取比例因子，以使之成为可用BCD格式表示的数值。为了对该数值取比因例子，首先必须确定它的大小，在取好比例因子之后，就要检查一下，看看结果是否落在所期望的范围之内，倘若答案为否定的话，即说明结果超出了给定的表示范围，那就表示需要对结果进行调整，根据情况左移(乘10)或右移(除10)一个10进位。由于在取比例因子的过程中，不可避免地带来了一些误差，所以，这样一个检查测试过程是必要的。

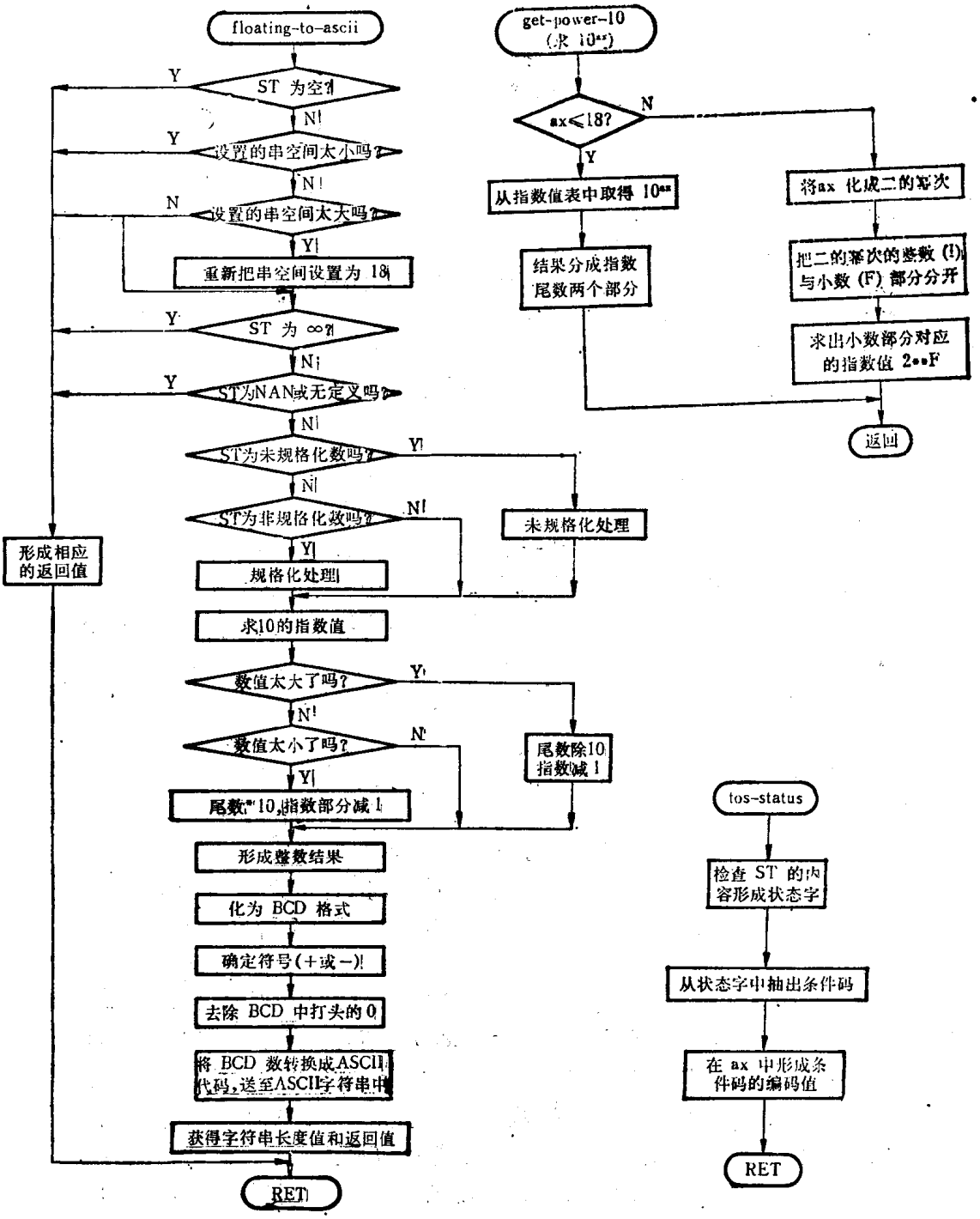
因为数值大小的估计只需要是近似的，故采用了一种快速技术，即用二的幂与之相乘，对应的浮点数的10进制指数将产生一个$\log_{10}2$倍的增量(因为$I*10^{x}*2^{k}=I*10^{x}*10^{\log_{10}2^{k}}=I*10^{x+k\cdot\log_{10}2}$)，再将该结果舍入成整数将得到一个足够精度的估计值。根据数值的大小以及数字字符串的规模就能够确定比例因子，比例因子的计算是整个转换过程中最不精确的操作，在此过程中，我们使用了$10^{x}=2**(x*\log_2 10)$这个关系，用到F2XM1这条指令。由于指令F2XM1所允许的自变量取值范围受到限制，因而必须把二的幂分成整数和小数两个部分，因为有关系2**(I+F)=2**I*2**F，所以，我们可以用F2XM1计算出2**F，然后，再用FSCALE指令将2**F和2**I结合起来，得到所需要的结果2**(I+F)。

为了避免在计算比例因子时发生上溢和下溢，我们把数值的小数部分和指数部分分开考虑。例如，若要将10^{-4932}换算到10^{18}，则需要比例因子10^{4950}，而它是8087所不能表示的(上溢)，但分开之后就不存在这一问题了，因为4950总是8087所能够表示的吧。在将指数部分与小数部分分开之后，取比例因子过程也就分成对应的两步，即指数部分相加，小数部分相乘，其指数运算中的操作数只包括一些较小的整数，因而，它们是8087所容易表示的。

此外，还存在这种可能性，幂函数(Get_Power_10)产生了一个比例因子，但用此比例因子之后得到的结果是ASCII字符串所无法表示的，即结果值太小。例如，对9.9999 9999 9999 9999 e4900取比例因子1.0000000000000009e-4883之后，就将得到结果1.0000000000000009e18，这里的比例因子是NDP所能够精确地表示的，得到的结果也说明其转换是精确的，但它却无法用BCD格式来表示。这也就是对结果值进行换算后测试的理由所在。根据结果的取值情况，对于数值太大或太小的结果，分别用10除或用10乘。

为了获得输出格式的最大灵活性，结果中小数点的位置由一个称为指数值的2进制整数指出。如果该值为0，就表示小数点在结果最低位的右面；若它大于0，就表示在结果的后面还要加上一些0，指数值指出了需添上的0的个数；倘若指数值小于0，就要将小数点在BCD串中左移，左移的位数由指数值规定。

转换过程的最后一步就是以BCD的形式存放结果，并指出结果中小数点的位置，最后，把BCD数字串分解成一些10进制数的ASCII字符。ASCII字符串中的符号对应于原始数值的符号(+或-)。




```
LINE   SOURCE
   1   $title(Convert a floating point number to ASCII)
   2           name  floating_to_ascii
   3           public floating_to_ascii
   4           extrn get_power_10:near, tos_status:near
   5   ;
```

```
6  ;       This subroutine will convert the floating point number in the
7  ; top of the 8087 stack to an ASCII string and separate power of 10
8  ; scaling value(in binary). The maximum width of the ASCII string
9  ; formed is controlled by a parameter which must be>1.Unnormal values,
10 ; denormal values,and psuedo zeroes will be correctly converted.
11 ; A returned value will indicate how many binary bits of
12 ; precision were lost in an unnormal or denormal value. The magnitude
13 ; (in terms of binary power)of a psuedo zero will also be indicated.
14 ; Integers less than 10**18 in magnitude are accurately converted if the
15 ; destination ASCII string field is wide enough to hold all the
16 ; digits. Otherwise the value is converted to scientific notation.  该子程序把 8087
     寄存器栈顶的浮点数转换成一个 ASCII 字符串和一个用 2 进制表示的十的幂。
17 ;
18 ;       The status of the conversion is identified by the return value,
19 ; it can be:  返回值所确定的一些转换结果状态:
20 ;
21 ;         0  conversion complete, string_size is defined      0: 转换完成
22 ;         1  invalid arguments                                1: 无效参数
23 ;         2  exact integer conversion, string_size is defined 2: 精确的整数转换
24 ;         3  indefinite                                       3: 无定义数值
25 ;         4  +NAN(Not A Number)                              4: +NAN
26 ;         5  -NAN                                            5: -NAN
27 ;         6  +Infinity                                       6: +∞
28 ;         7  -Infinity                                       7: -∞
29 ;         8  psuedo zero found, string_size is defined       8: 发现伪 0
30 ;
31 ;       The PLM/86 calling convention is:
32 ;
33 floating_to_ascii:
34 ; procedure(number, denormal_ptr, string_ptr, size_ptr, field_size,
35 ;          power_ptr)word external;
36 ; declare (denormal_ptr, string_ptr, power_ptr, size_ptr)pointer;
37 ; declare field_size word, string_size based size_ptr word;
38 ; declare number real;
39 ; declare denormal integer based denormal_ptr;
40 ; declare power integer based power_ptr;
41 ; end floating_to_ascii;
42 ;
43 ;       The floating point value is expected to be on the top of the NPX
```

```
44  ; stack. This subroutine expects 3 free entries on the NPX stack and
45  ; will pop the passed value off when done. The generated ASCII string
46  ; will have a leading character either '-' or '+' indicating the sign
47  ; of the value. The ASCII decimal digits will immediately follow.
48  ; The numeric value of the ASCII string is (ASCII STRING.)*10**POWER. 该子
    程序需要 8087 堆栈中的三个单元，结束后它将如数归还。产生的 ASCII 串以"-"
    或"+"开头，后接 ASCII 10 进制数字，其数值为：(ASCII STRING.)*10**POWER。
49  ; It the given number was zero, the ASCII string will contain a sign
50  ; and a single zero chacter. The value string_size indicates the total
51  ; length of the ASCII string including the sign character. string(0)will
52  ; always hold the sign. It is possible for string_size to be less than
53  ; field_size. This occurs for zeroes or integer values. A psuedo zero
54  ; will return a special return code. The denormal count will indicate
55  ; the power of two originally associated with the value. The power of
56  ; ten and ASCII string will be as if the value was an ordinary zero.  string_size 指
    出 ASCII 串的长度，string(0)为符号。string_size 有可能小于 field_size，对于 0
    或整数值，更会出现这种情况。十的幂和 ASCII 字符串的初值为 0。
57  ;
58  ;       This subroutine is accurate up to a maximum of 18 decimal digits for
59  ; integers. Integer values will have a decimal power of zero associated
60  ; with them. For non integers, the result will be accurate to within 2
61  ; decimal digits of the 16th decimal place (double precision).The
62  ; exponentiate instruction is also used for scaling the value into the
63  ; range acceptable for the BCD data type. The rounding mode in effect
64  ; on entry to the subroutine is used for the conversion.
65  ;
66  ;       The following registers are not transparent,
67  ;
68  ;                       ax bx cx dx si di flags
69  ;
70
71  ;
72  ;       Define the stack layout.  规定栈的格式
73  ;
74  bp_save         equ   word ptr [bp]
75  es_save         equ   bp_save + size bp_save
76  return_ptr      equ   es_save + size es_save
77  power_ptr       equ   return_ptr + size return_ptr
78  field_size      equ   power_ptr + size power_ptr
```

```
 79  size__ptr          equ  field__size + size field__size
 80  string__ptr        equ  size__ptr + size size__ptr
 81  denormal__ptr      equ  string__ptr + size string__ptr
 82
 83  parms__size        equ  size power__ptr + size field__size + size size__ptr +
 84  &                       size string__ptr + size denormal__ptr
 85  ;
 86  ;       Define constants used  定义常数;
 87  ;
 88  BCD__DIGITS        equ  18  ; Number of digits in bcd__value  BCD__DIGITS=18,
                                    表示 bcd__value 可用 18 位表示.
 89  WORD__SIZE         equ  2
 90  BCD__SIZE          equ  10
 91  MINUS              equ  1   ; Define return values  这里主要是定义返回值所代表的
                                    意义.
 92  NAN                equ  4   ; The exact values chosen here are
 93  INFINITY           equ  6   ; important. They must correspond to
 94  INDEFINITE         equ  3   ; the possible return values and be in
 95  PSUEDO__ZERO       equ  8   ; the same numeric order as tested by
 96  INVALID            equ  -2  ; the program.
 97  ZERO               equ  -4
 98  DENORMAL           equ  -6
 99  UNNORMAL           equ  -8
100  NORMAL             equ  0
101  EXACT              equ  2
102  ;
103  ;       Define layout of temporary storage area.  定义暂存区域的格式
104  ;
105  status             equ  word ptr [bp__WORD__SIZE]
106  power__two         equ  status__WORD__SIZE
107  power__ten         equ  power__two__WORD__SIZE
108  bcd__value         equ  tbyte ptr power__ten__BCD__SIZE
109  bcd__byte          equ  byte ptr bcd__value
110  fraction           equ  bcd__value
111
112  local__size        equ  size status + size power__two + size power__ten
113  &                       + size bcd__value
114  ;
115  ;       Allocate stack space for the temporaries so the stack will be big enough  分配
```

```
        临时栈空间,以使栈空间足够大
116 ;
117 stack           segment stack 'stack'
118                 db      (local_size + 6)dup(?)
119 stack           ends
120
121 cgroup          group code
122 code            segment public 'code'
123                 assume cs:cgroup
124                 extrn   power_table:qword
125 ;
126 ;       Constants used by this function.
127 ;
128                 even            ; Optimize for 16 bits
                                    规定从偶地址开始分配存贮器,以优化字的存取
129 const10         dw      10      ; Adjustment value for too big BCD
                                    用作太大的BCD数的修正值
130 ;
131 ;       Convert the C3,C2,C1,C0 encoding from tos_status into meaningful bit
132 ; flags and values. 从子程序 tos_status 所获得的条件码 C3、C2、C1、C0 所代表的
        状态
133 ;
134 status_table    db      UNNORMAL, NAN, UNNORMAL + MINUS, NAN +
                            MINUS,
135 &                       NORMAL, INFINITY, NORMAL + MINUS, INFINITY +
                            MINUS,
136 &                       ZERO, INVALID, ZERO + MINUS, INVALID,
137 &                       DENORMAL, INVALID, DENORMAL + MINUS, INVA-
                            LID
138
139 floating_to_ascii proc
140
141     call    tos_status          ; Look at status of ST(0)  检查栈顶 ST(0) 的状态
142     mov     bx, ax              ; Get descriptor from table
                                    从状态表中取得状态描述信息
143     mov     al, status_table[bx]
144     cmp     al, INVALID         ; Look for empty ST(0)  ST(0)为空吗?
145     jne     not_empty
146 ;
```

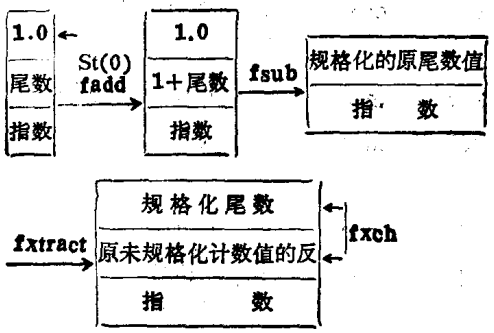
```
147 ;       ST(0) is empty; Return the status value.  ST(0)为空,返回状态值
148 ;
149   ret     parms__size
150 ;
151 ;       Remove infinity from stack and exit.  从栈中移去无穷大,返回
152 ;
153 found__infinity:
154
155   fstp    st(0)                    ; OK to leave fstp running
156   jmp     short exit__proc
157 ;
158 ;       string space is too small;Return invalid code.  字符串空间太小,返回无效代码
159 ;
160 small__string:
161
162   mov     al, INVALID
163
164 exit__proc:
165
166   mov     sp, bp                   ; Free stack space  释放栈空间
167   pop     bp                       ; Restore registers  恢复寄存器的内容
168   pop     es
169   ret     parms__size
170 ;
171 ;       ST(0)is NAN or indefinite. Store the value in memory and look
      ST(0)中是NAN或无定义的代码,此时先将它送至内存,再检查其小数部分,以将
      NAN与无定义代码分开
172 ; at the fraction field to separate indefinite from an ordinary NAN.
173 ;
174 NAN__or__indefinite:
175
176   fstp    fraction                 ; Remove value from stack for examination
                                       从栈顶移出该值,以便检查
177   test    al, MINUS                ; Look at sign bit  看符号位
178   fwait                            ; Insure store is done  fwait指令可以保证fstp
                                       (存数)完成后再继续执行
179   jz      exit__proc               ; Can't be indefinde if positive  若符号为正,
                                       则表示它是NAN.
180
```

```
181     mov     bx, 0c000H              ; Match against upper 16 bits of fraction
182     sub     bx, word ptr fraction+6 ; Compare bits 63—48 将小数部分的高 16
                                          位(63～48)与 0C000H 比较；小数的其余部
                                          分必须是 0
183     or      bx, word ptr fraction+4 ; Bits 32—47 must be zero
184     or      bx, word ptr fraction+2 ; Bits 31—16 must be zero
185     or      bx, word ptr fraction   ; Bits 15—0 must be zero
186     jnz     exit_proc
187
188     mov     al, INDEFINITE          ; Set return value for indefinite value 为该
                                          无定义值设置返回值，返回
189     jmp     exit_proc
190 ;
191 ;       Allocate stack space for local variables and establish parameter
192 ; addressibility. 为局部变量分配栈空间，且建立参数的寻址机构
193 ;
194 not_empty:
195
196     push    es                      ; Save working register 保护工作寄存器 es、
                                          bp
197     push    bp
198     mov     bp, sp                  ; Establish stack addressibility 建立栈的寻
                                          址机构
199     sub     sp, local_size
200
201     mov     cx, field_size          ; Check for enough string space 检查字符串
                                          空间是否足够。当它不到两个时，说明太小
                                          了。
202     cmp     cx, 2
203     jl      small_string
204
205     dec     cx                      ; Adjust for sign character 符号占用了一
                                          个字符空间
206     cmp     cx, BCD_DIGITS          ; See if string is too large for BCD 看字
                                          符串空间是否太大(>18)?
207     jbe     size_ok
208
209     mov     cx, BCD_DIGITS          ; Else set maximum string size 太大，则
                                          重新规定最大串长度 =18（因为 BCD 数
```

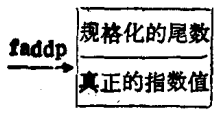
不会超过 18 位)

```
210
211 size__ok:
212
213     cmp     al, INFINITY            ; Look for infinity  8087栈顶内容为无穷大吗?
214     jge     found__infinity         ; Return status value for + or - inf.
                                        是,返回 +∞ 或 -∞ 的状态码
215
216     cmp     al, NAN                 ; Look for NAN or INDEFINITE  检查ST(0)是否为NAN或无定义
217     jge     NAN__or__indefinite
218 ;
219 ;       Set default return values and check that the number is normalized.
220 ;
221     fabs                            ; Use positive value only
222                                     ; sign bit in al has true sign of value  只用正数,真正的符号保留在al中
223     mov     dx, ax                  ; save return value for later  保护返回值
224     xor     ax, ax                  ; Form 0 constant
225     mov     di, denormal__ptr       ; zero denormal count  置0未规格化计数值
226     mov     word ptr[di],ak
227     mov     bx, power__ptr          ; zero power of ten value  将十的幂值也置0
228     mov     word ptr[bx], ax
229     cmp     dl, ZERO                ; Test for zero  检查待转换的浮点数是否为0
230     jae     real__zero              ; skip power code if value is zero
231
232     cmp     dl, DENORMAL            ; Look for a denormal value  对未规格化数进行专门处理
233     jae     found__denormal         ; Handle it specially
234
235     fxtract                         ; Separate exponent from significand  将指数部分与有效位分开
236     cmp     dl, UNNORMAL            ; Test for unnormal value  非规格化的待转换浮点数吗?
237     jb      normal__value
```

```
238
239     sub     dl, UNNORMAL__NORMAL  ; Return normal status with correct sign
                                      设置规格化状态位
240     ;
241     ;       Normalize the fraction, adjust the power of two in ST (1) and set
242     ; the denormal count value.  规格化小数部分，修改ST(1) 中二的指数值且设置未
        规格化计数值，以使 0<=ST(0)<1.0
243     ;
244     ; Assert: 0<=ST(0)<1.0
245     ;
246     fldl                          ; Load constant to normalize fraction
247
248 normalize__fraction,
249
250     fadd    st(1), st             ; Set integer bit in fraction
251     fsub                          ; Form normalized fraction in ST(0)
252     fxtract                       ; Power of two field will be negative
253                                   ; of denormal count
254     fxch                          ; Put denormal count in ST(0)
                                      常数 1 压入 8087 堆栈
```

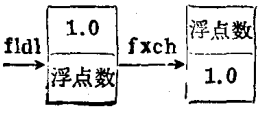


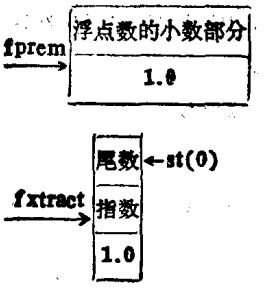

```
255     fist    word ptr [di]         ; put negative of denormal count in memory
256     faddp   st(2), st             ; Form correct power of two in st(1)
```




```
257                                   ; OK to use word ptr [di] now
258     neg     word ptr[di]          ; Form positive denormal count
259     jnz     not__psuedo__zero
260     ;
261     ;       A psuedo zero will appear as an unnormal number. when attempting
262     ; to normalize it, the resultant fraction field will be zero. performing
```

```
263   ; an fxtract on zero will yield a zero exponent value.
264   ;
265     fxch                                  ; put power of two value in st(0)
                                                指数值⇒st(0)
266     fistp     word ptr [di]               ; Set denormal count to power of two
                                                value 设置未规格化计数值
267                                           ; word ptr [di] is not used by convert
268                                           ; integer, OK to leave running
269     sub       dl, NORMAL-PSUEDO_ZERO      ; Set return value saving the sign bit
                                                设置返回值,保护符号位
270     jmp       convert_integer             ; Put zero value into memory
271   ;
272   ;       The number is a real zero, set the return value and setup for
              待转换的数是一个真 0,设置返回值,开始到 BCD 的转换
273   ; conversion to BCD.
274   ;
275   real_zero:
276
277     sub       dl, ZERO-NORMAL             ; Convert status to normal value
                                                把状态置为规格化的值
278     jmp       convert_integer             ; Treat the zero as an integer
                                                把 0 当作整数处理
279   ;
280   ;       The number is a denormal. FXTRACT will not work correctly in this
              待转换的数是未规格化数,此时 FXTRACT 不能正确执行,故做如下处理:
281   ; case. To correctly separate the exponent and fraction, add a fixed
282   ; constant to the exponent to guarantee the result is not a denormal.
283   ;
284   found_denormal:
285
286     fld1                                  ; prepare to bump exponent
287     fxch
288     fprem                                 ; Force denormal to smallest repr-
                                                esentable
289                                           ; extended real format exponent
290     fxtract                               ; This will work correctly now
```






```
291  ;
292  ;       The power of the original denormal value has been safely isolated.
293  ; Check if the fraction value is an unnormal.
     至此,原未规格化数的幂已被安全隔离. 看小数是否非规格化
294  ;
295     fxam                          ; See if the fraction is an unnormal
296     fstsw    status               ; Save status for later
297     fxch                          ; Put exponent in ST (0)
                                      检测小数部分,得到并保护状态字
298     fxch     st(2)                ; Put 1.0 into ST(0), exponent in
                                      ST(2)  1.0⇒st(0),尾数⇒st(1),
                                      指数⇒st(2)
299     sub      dl, DENORMAL-NORMAL  ; Return normal status with correct
                                      sign
                                      得到规格化的返回值,符号位不变
300     test     status, 4400H        ; See if C3 = C2 = 0 impling unnor-
                                      mal or NAN  若C3=C2=0, 则
                                      表示尾数是NAN或非规格化的
301     jz       normalize_fraction   ; Jump if fraction is an unnormal
302
303     fstp     st(0)                ; Remove unnecessary 1.0 from
                                      st(0)  1.0退栈,此时栈顶为
                                                  尾数 ←st(0)
                                                  指数
304  ;
305  ;       Calculate the decimal magnitude associated with this number to
306  ; within one order. This error will always be inevitable due to
307  ; rounding and lost precision. As a result, we will deliberately fail
308  ; to consider the LOG10 of the fraction value in calculating the order.
309  ; Since the fraction will always be 1<=F<2, its LOG10 will not change
310  ; the basic accuracy of the function. To get the decimal order of magnitude,
311  ; simply multiply the power of two by LOG10(2) and truncate the result to
```

```
312   ; an integer.  计算对应的 10 进制数值，由于舍入及精度丢失，误差是不可避免的。
        为了获得 10 进制阶数，只要将 2 的幂次与 log10(2) 相乘，且将结果舍入成整数即可
313   ;
314   normal_value:
315   not_psuedo_zero:
316
317     fstp    fraction                ; Save the fraction field for later use
                                          保护尾数部分，以便后面使用。尾数 = > fraction
318     fist    power_two               ; Save power of two
                                          2 的幂 = > power_two
319     fldlg2                          ; Get LOG10(2)  st(0) < = log10(2)
320                                     ; Power_two is now safe to use
321     fmul                            ; Form LOG 10 (of exponent of number)
                                          st(0) < = 指数*log10(2)
322     fistp   power_ten               ; Any rounding mode will work here
                                          st(0) = > power_ten；power_ten 中含有
                                          10 的整数次方
323   ;
324   ;       Check if the magnitude of the number rules out treating it as  检查数值太小。
325   ; an integer.
326   ;
327   ; CX has the maximum number of decimal digits allowed.
        CX 中存放着允许的 10 进制数的最多位数
328   ;
329     fwait                           ; wait for power_ten to be valid
                                          数据同步，等待 fistp 结束
330     mov     ax, power_ten           ; Get power of ten of value
331     sub     ax, cx                  ; Form scaling factor necessary in ax
                                          在 ax 中形成必要的比例因子
332     ja      adjust_result           ; Jump if number will not fit
                                          若数值过大，则跳到 adjust_result
333   ;
334   ;       The number is between 1 and 10** (field_size).
        否则，表示它在 1 到 10**(field_size)之间，看它是否为整数
335   ; Test if it is an integer.
336   ;
337     fild    power_two               ; Restore original number
338     mov     si, dx                  ; Save return value  保护返回值
```

```
339     sub     dl, NORMAL-EXACT        ; Convert to exact return value
                                        把返回值置为精确的整数转换
340     fld     fraction
341     fscale                          ; Form full value,  this is safe here
                                        形成完整的原始值
342     fst     st(1)                   ; Copy value for compare
                                        复制该值,以便比较
343     frndint                         ; Test if its an integer
                                        将原值与其整数值比较,看是否相等
344     fcomp                           ; Compare values
345     fstsw   status                  ; Save status
346     test    status, 4000H           ; C3 = 1 implies it  was an integer
                                        C3 = 1 时,表示待转换的是一个整数
347     jnz     convert_integer
348
349     fstp    st(0)                   ; Remove non integer value
350     mov     dx, si                  ; Re store original return value
                                        否则,恢复原来的返回值
351 ;
352 ;       Scale the number to within the range allowed by the BCD format.
        为待转换的数取比例因子,以使其可以用 BCD 格式表示
353 ; The scaling operation should produce a number within one decimal order
354 ; of magnitude of the largest decimal number representable within the
355 ; given string width.
356 ;
357 ;       The scaling power of ten value is in ax.  比例因子(十的幂值)在寄存器 ax 中
358 ;
359 adjust_result;
360
361     mov     word ptr [bx],ax        ; Set initial power of ten return value
362     neg     ax                      ; Subtract one for each order of
                                        每对数值取一阶比例因子,就要从 ax 中
                                        减去 1(+10)
363                                     ; magnitude the value is scaled by
364     call    get_power_10            ; Scaling factor is returned as exponent
                                        以指数和尾数的形式返回 10**ax
365                                     ; and fraction
366     fld     fraction                ; Get fraction
367     fmul                            ; Combine fractions  取回原小数值,将它
```

```
                                        与上面的尾数相乘
368     mov     si, cx                  ; Form power of ten of the maximum
                                        形成数值串能表示的最大的BCD数的
                                        10的幂次,索引在si中
369     shl     si, 1                   ; BCD value to fit in the string
370     shl     si, 1                   ; Index in si
371     shl     si, 1
372     fild    power_two               ; Combine powers of two
                                        将2的指数加起来,形成原数值的指数
373     faddp   st(2), st
374     fscale                          ; Form full value, exponent was safe
                                        形成完整的数值
375     fstp    st(1)                   ; Remove exponent
376 ;
377 ;       Test the adjusted value against a table of exact powers of ten.
378 ; The combined errors of the magnitude estimate and power function can
379 ; result in a value one order of magnitude too small or too large to fit
380 ; correctly in the BCD field. To handle this problem, pretest the
381 ; adjusted value, if it is too small or large, then adjust it by ten and
382 ; adjust the power of ten value. 对上面修正过的值进行检查，若其值太大或太小，
    不能正好装满BCD字段,则再次用10(*或÷)对之修正,并修改10的指数值
383 ;
384 test_power:
385
386     fcom    power_table[si] + type power_table; Compare against exact power
                                        比较,看数值是否太大
387                                     ; entry. Use the next entry since cx
388                                     ; has been decremented by one
389     fstsw   status                  ; No wait is necessary
                                        不需等待指令,fstsw具有指令同步功能
390     test    status, 4100H           ; If C3=C0=0 then too big
                                        若C3=C0=0,则表示该数值太大
391     jnz     test_for_small
392
393     fidiv   const10                 ; Else adjust value  修正结果(+10)
394     and     dl, not EXACT           ; Remove exact flag  去除EXACT(精确)
                                        标志
395     inc     word ptr[bx]            ; Adjust power of ten value
                                        修改10的指数值(+1)
```

```
396     jmp     short in_range          ; Convert the value to a BCD integer
397
398 test_for_small:
399
400     fcom    power_table[si]         ; Test relative size 比较,看数值是否太小
401     fstsw   status                  ; No wait is necessary
402     test    status, 100H            ; It C0 = 0 then st(0) > = lower bound
                                        若 C0 = 0, 则表示该数在其表示范围之内
403     jz      in_range                ; Convert the value to a BCD integer
404
405     fimul   const 10                ; Adjust value into range
                                        否则,即该值太小,对之进行修正(*10)
406     dec     word ptr[bx]            ; Adjust power of ten value
                                        修改 10 的指数值(-1)
407
408 in_range:
409
410     frndint                         ; Form integer value 形成整数值
411 ;
412 ; Assert: 0 < = TOS < = 999, 999, 999, 999, 999, 999
    0 < = ST < = 999,999,999,999,999,999
413 ; The TOS number will be exactly representable in 18 digit BCD format.
    故栈顶内容可以精确地用 BCD 格式表示
414 ;
415 convert_integer:
416
417     fbstp   bcd_value               ; Store as BCD format number 将该整数
                                        以 BCD 数的格式存放到 bcd_value 中
418 ;
419 ;       while the store BCD runs, setup registers for the conversion to
    在存放 BCD 数的同时,建立 BCD→ASCII 转换过程中用到的寄存器的状态
420 ; ASCII.
421 ;
422     mov     si, BCD_SIZE_2          ; Initial BCD index value
                                        初始化 BCD 的索引
423     mov     cx, 0f04h               ; Set shift count and mask 设置移位计数
                                        值(4)及 BCD 的位抽取值(0FH)
424     mov     bx, 1                   ; Set initial size of ASCII field for sion
                                        为符号先分配一个 ASCII 字符空间
```

```
425     mov     di, string__ptr          ; Get address of start of ASCII string
                                           取 ASCII 字符串的超始地址
426     mov     ax, ds                   ; Copy ds to es
427     mov     es, ax
428     cld                              ; Set autoincrement mode
                                           规定为递减方式
429     mov     al, '+'                  ; Clear sign field  确定符号"+"或"-"
430     test    dl, MINUS                ; Look for negative value
431     jz      positive__result
432
433     mov     al,'-'
434
435     Positive__result:
436
437     stosb                            ; Bump string pointer past sign
                                           把符号送到 ASCII 串
438     and     dl, not MINUS            ; Turn off sign bit  去掉符号位
439     fwait                            ; wait for fbstp to finish
                                           等待指令 fbstp(417 号)完成
440 ;
441 ;       Register usage:  寄存器应用说明:
442 ;                                ah:    BCD byte value in use
                                           ah: 使用中的 BCD 字节值
443 ;                                al:    ASCII character value
                                           al: ASCII 字符值
444 ;                                dx:    Return value  dx: 返回值
445 ;                                ch:    BCD mask = 0fh
                                           ch: BCD 位抽取值 = 0 FH
446 ;                                cl:    BCD shift count = 4
                                           cl: BCD 移位计数值 = 4
447 ;                                bx:    ASCII string field width
                                           bx: ASCII 字符串的长度
448 ;                                si:    BCD field index  si: BCD 的索引
449 ;                                di:    ASCII string field pointer
                                           di: ASCII 字符串的指针
450 ;                                ds, es: ASCII string segment base
                                           ds, es: ASCII字符串的段基址
451 ;
452 ;       Remove leading zeroes from the number.  去除数值头上的一些 0
```

```
453  ;
454  skip_leading_zeroes:
455
456      mov     ah, bcd_byte[si]        ; Get BCD byte  取 BCD 字节值 = >ah
457      mov     al, ah                  ; Copy value  al< = ah
458      shr     al, cl                  ; Get high order digit  取高位 BCD 数
459      and     al, ch                  ; Set zero flag
460      jnz     enter_odd               ; Exit loop if leading non zero found
                                           若发现它不是 0,则转出该"去头 0"循环
461
462      mov     al, ah                  ; Get BCD byte again
                                           再取回该 BCD 字节值 al< = ah
463      and     al, ch                  ; Get low order digit  取低位 BCD 数
464      jnz     enter_even              ; Exit loop if non zero digit found
                                           若低位 BCD 数不是 0,也跳出循环
465
466      dec     si                      ; Decrement BCD index
                                           否则,表示该 BCD 字节为 0,BCD 索引值
                                           减 1,继续去除 BCD 中打头的 0
467      jns     skip_leading_zeroes
468  ;
469  ;       The significand was all zeroes.  若执行到这里,就说明 BCD 数为全 0
470  ;
471      mov     al, '0'                 ; Set initial zero
                                           送字符"0"到 ASCII 字符串
472      stosb
473      inc     bx                      ; Bump string length
                                           ASCII 字符串的长度加 1
474      jmp     short exit_with_value   跳转至 exit_with_value, 进行结束处理
475  ;
476  ;       NOW expand the BCD string into digit per byte values 0~9.
         把 BCD 数字串分开,使得每个字节的值为 0~9 的 ASCII 码
477  ;
478  digit_loop:
479
480      mov     ah, bcd_byte[si]        ; Get BCD byte  取 BCD 字节
481      mov     al, ah
482      shr     al, cl                  ; Get high order digit  取高位 BCD 数
483      and     al. ch
```

```
484  enter__odd,
485
486    add      al,'0'                ; Convert to ASCII  将该 BCD 位转换成
                                        ASCII 码
487    stosb                          ; Put digit into ASCII string area  把此数
                                        字送至 ASCII 字符串中
488    mov      al, ah                ; Get low order digit  取 BCD 字节的低位
                                        BCD 数
489    and      al, ch
490    inc      bx                    ; Bump field size counter  ASCII 字符串
                                        长度增 1
491
492  enter__even,
493
494    add      al, '0'               ; Convert to ASCII  将低位 BCD 数化为
                                        对应的 ASCII 码
495    stosb                          ; Put digit into ASCII area  把该数字送
                                        到 ASCII 字符串区域
496    inc      bx                    ; Bump field size counter  ASCII 字符串
                                        长度增 1
497    dec      si                    ; Go to next BCD byte  继续处理下一个
                                        BCD 字节,直至完成
498    jns      digit__loop
499  ;
500  ;      Conversion complete. set the string size and remainder.  至此转换过程结束,
       设置字符串长度
501  ;
502  exit__with__value,
503
504    mov      di, size__ptr
505    mov      word ptr [di], bx
506    mov      ax, dx                ; Set return value  设置返回值
507    jmp      exit__proc
508
509  floating__to__ascii    endp
510  code                   ends
511                         end
```

ASSEMBLY COMPLETE, NO ERRORS FOUND

```
LINE  SOURCE
   1  $title(Calculate the value of 10**ax)
   2  ;
   3  ;       This subroutine will calculate the value of 10**ax.
   4  ; All 8086 registers are transparent and the value is returned on
   5  ; the TOS as two numbers, exponent in ST(1) and fraction in ST(0).
   6  ; The exponent value can be larger than the maximum representable
   7  ; exponent. Three stack entries are used.  该子程序计算 10**ax 的值,返回值是放在
        数值栈顶的两个数,即放在 ST(1)的指数部分及放在 ST(0)的尾数部分
   8  ;
   9                      name    get_power_10
  10                      public  get_power_10, power_table
  11
  12  stack               segment stack 'stack'
  13                      dw      4 dup (?)          ; Allocate space on the stack
                                                       分配栈空间
  14  stack               ends
  15
  16  cgroup              group   code
  17  code                segment public 'code'
  18                      assume  cs:cgroup
  19  ;
  20  ;       Use exact values from 1.0 to 1e18.  1.0 到 10^18(1e18)的精确的指数值表
  21  ;
  22                      even                       ; Optimize 16 bit access 规
                                                       定始址为偶数，以优化字
                                                       的存取
  23  power_table         dq      1.0, 1e1, 1e2, 1e3
  24                      dq      1e4, 1e5, 1e6, 1e7
  25                      dq      1e8, 1e9, 1e10, 1e11
  26                      dq      1e12, 1e13, 1e14, 1e15
  27                      dq      1e16, 1e17, 1e18
  28
  29  get_power_10  proc
  30
  31    cmp       ax, 18                           ; Test for 0 <= ax < 19 看
                                                       ax 是否满足 0 <= ax < 19
  32    ja        out_of_range  若 ax >= 19, 则表示该数太大,超出表示范围
  33
```

```
34    push    bx                  ; Get working index register  保护 bx，将
                                    bx 用作变址寄存器
35    mov     bx, ax              ; Form table index  形成 power__table 表
                                    的索引地址
36    shl     bx, 1
37    shl     bx, 1
38    shl     bx, 1
39    fld     power__table[bx]    ; Get exact value  得到精确的10的指数值
40    pop     bx                  ; Restore register value  恢复寄存器 bx 的
                                    内容
41    fxtract                     ; Separate power and fraction  将所形成
                                    的值的指数和尾数部分分开
42    ret                         ; OK to leave fxtract running  返回，让指
                                    令 fxtract 继续运行
43  ;
44  ;       Calculate the value using the exponentiate instruction.
45  ; The following relations are used:
46  ;                           10**x = 2**(log2(10)*x)
47  ;                           2**(I+F) = 2**I*2**F
    利用指数运算指令及下列关系计算 10**ax:
                                10**x = 2**(log2 10*x)
                                2**(I+F) = 2**I*2**F
48  ; if st(1) = I and st(0) = 2**F then fscale produces 2**(I+F)  若 st(1) = I 及 st(0)
    = 2**F，则 fscale 产生 2**(I+F)
49  ;
50  out__of__range:
51
52    fld12t                      ; TOS = LOG2(10)  ST< = log2 10
53    push    bp                  ; Establish stack addressibility  建立栈的
                                    寻址机构
54    mov     bp, sp
55    push    ax                  ; put power (P) in memory  将指数(P)送
                                    到存贮器中
56    push    ax                  ; Allocate space for status  为 8087 的控制
                                    字分配空间
57    fimul   word ptr [bp-2]     ; TOS, x = LOG2 (10) *p = LOG2 (10**p)
                                    ST, X = log2 10*p = log2 (10**p)
58    fnstcw  word ptr [bp-4)     ; Get current control word  保护当前控制
                                    字
```

```
59                                      ; Control word is a static value
60      mov     ax, word ptr [bp-4]     ; Get control word, no wait necessary  取得当
                                          前控制字,不需要等待
61      and     ax, not 0C00H           ; Mask off current rounding field  把舍入方式
                                          设置为向下舍入(趋向-∞)
62      or      ax, 0400H               ; Set round to negative infinity
63      xchg    ax, word ptr [bp-4]     ; Put new control word in memory
64                                      ; old control word is in ax
65      fldl                            ; Set TOS=-1.0  置ST=-1.0
66      fchs
67      fld     st(1)                   ; Copy power value in base two  对应的2的
                                          指数值送至栈顶(X)
68      fldcw   word ptr [bp-4]         ; Set new control word value  设置新控制字
                                          (向下舍入)
69      frndint                         ; TOS=I, -inf <I<=x, I is an integer  取
                                          整; ST=I, -∞<I<=X, I为整数
70      mov     word ptr [bp-4], ax     ; Restore original rounding control  恢复原控
                                          制字的舍入方式
71      fldow   word ptr [bp-4]
72      fxch    st(2)                   ; TOS=x, ST(1) =-1.0, ST(2)=I  ST=X,
                                          ST(1)=-1.0, ST(2)=I
73      pop     ax                      ; Remove original control word  移去原控制字
74      fsub    st, st(2)               ; TOS, F=X-I, 0<=TOS<1.0  ST, F=X
                                          -I, 0<=ST<1.0
75      pop     ax                      ; Restore power of ten  恢复10的指数值,即
                                          ax<=10的指数
76      fscale                          ; TOS=F/2, 0<=TOS<0.5  ST=F*2^-1.0=
                                          F/2, 0<=ST<0.5
77      f2xml                           ; TOS=2** (F/2) -1.0
78      pop     bp                      ; Restore stack
79      fsubr                           ; Form 2**(F/2)  2**(F/2)-1.0-(-1.0)=
                                          2**(F/2)
80      fmul    st, st(0)               ; Form 2**F  2**(F/2)**2** (F/2)=2**F
81      ret                             ;  to leave fmul running
82      至此,10^ax 也被分成指数和尾数两部分; ST—尾数  ST(1)—指数
83  get_power_10  endp
84  code          ends
85                end
ASSEMBLY COMPLETE, NO ERRORS FOUND
```

```
LINE  SOURCE
   1  $title(Determine TOS register contents)
   2  ;
   3  ;       This subroutine will return a value from 0—15 in ax corresponding
   4  ; to the contents of 8087 TOS. All registers are transparent and no
   5  ; errors are possible. The return value corresponds to C3, C2, C1, C0
   6  ; of FXAM instruction.  本子程序将根据 8087 栈顶寄存器的内容，在 ax 中返回一
        个 0～15之间的值,其返回值分别对应于指令 FXAM 所形成的条件码(C3, C2, C1,
        C0)
   7  ;
   8    name        tos_status
   9    public      tos_status
  10
  11  stack         segment stack 'stack'
  12                dw      3 dup (?)  ; Allocate space on the stack
                                         分配存贮器堆栈空间
  13  stack         ends
  14
  15  cgroup        group   code
  16  code          segment public 'code'
  17                assume  cs:cgroup
  18  tos_status    proc
  19
  20    fxam                           ; Get register contents status
                                         形成ST 中内容的状态
  21    push        ax                 ; Allocate space for status value
                                         为 8087 的状态字保留存贮空间
  22    push        bp                 ; Establish stack addressibility
                                         建立堆栈的寻址机构
  23    mov         bp, sp
  24    fstsw       word ptr [bp+2]    ; Put tos status in memory
                                         状态字送到存贮器
  25    pop         bp                 ; Restore registers
  26    pop         ax                 ; Get status value, no wait necessary
                                         取得状态字的内容,不要等待
  27    mov         al, ah             ; Put bit 10—8 into bits 2—0
  28    and         ax, 4007h          ; Mask out bits C3, C2, C1, C0
  29    shr         ah, 1              ; Put bit C3 into bit 11
  30    shr         ah, 1
```

```
31      shr         ah, 1
32      or          al, ah              ; Put C3 into bit 8
33      mov         ah, 0               ; Clear return value
34      ret
35
36  tos_status      endp
37  code            ends
38                  end
ASSEMBLY COMPLETE, NO ERRORS FOUND
```

例 4：

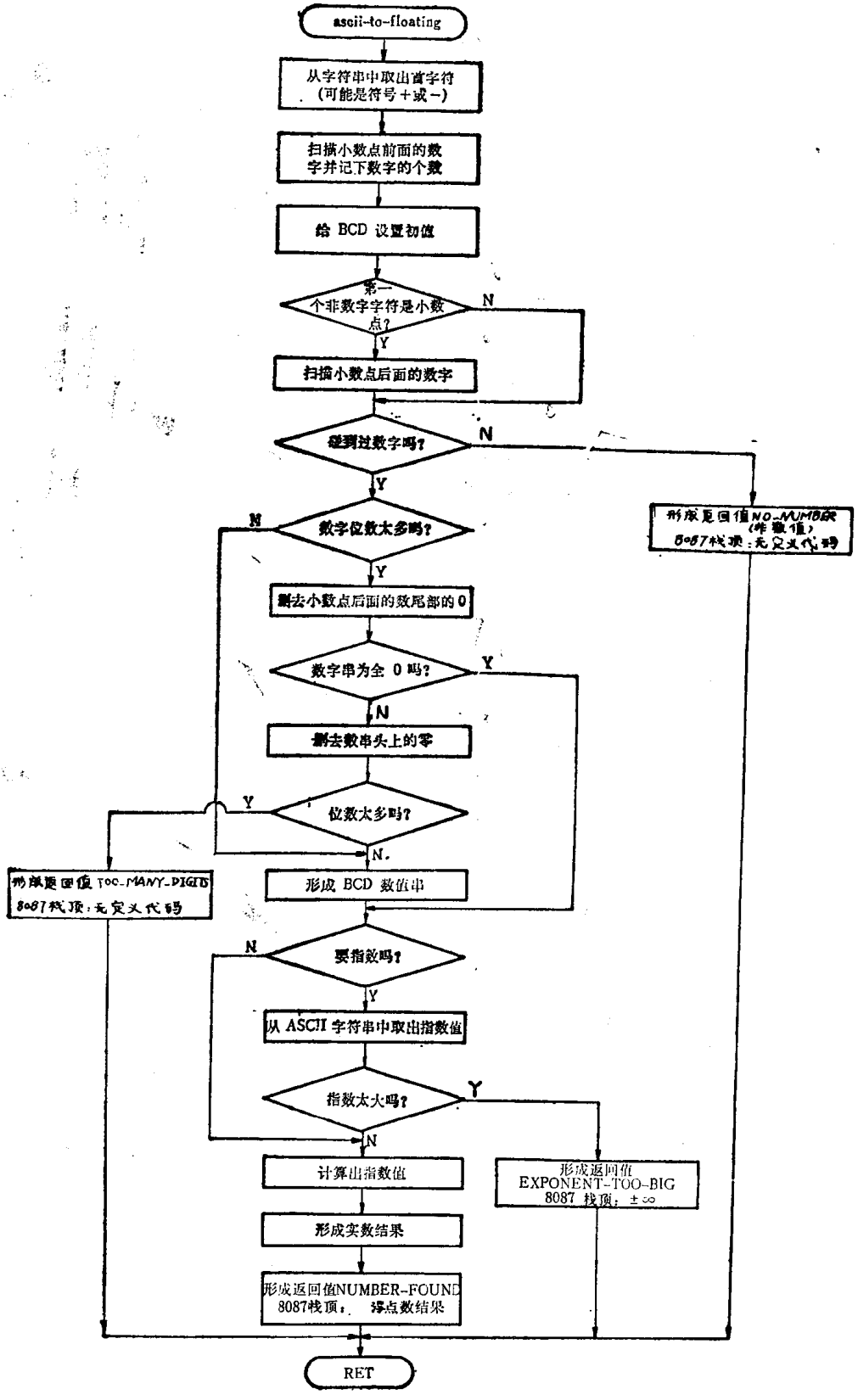
本例所起的作用刚好与例 3 相反，它是把 ASCII 输入字符串转换成相应的浮点数值，其返回值可以由 PLM/86、PASCAL/86、FORTRAN/86 及 ASM/86 程序所使用。该子程序接收的是以标准的 FORTRAN 格式所表示的 ASCII 数字串，一次最多可以接收 18 个 10 进制数位。

从数值运算的角度来讲，把 ASCII 转换成浮点数这一过程没有把浮点数转换成 ASCII 字符串那样复杂。我们可以将 ASCII 输入串转换为浮点数的过程分成四步：确定输入数据的 10 进制位数；若 10 进制小数点在最右面，则形成对应于该数值串的 BCD 值；计算指数值；获得 BCD 数值。前三步都由主机软件完成，而第四步则主要由 8087 的数值指令所实现。

由于输入数值的灵活性，从而增加了本函数(过程)的复杂性，从下面列出的源程序清单中可以看到，其中的大多数代码都用于确定各个字符的含义。程序还必须识别分开的两个输入数据：尾数值和指数值。至于数值运算到是比较简单的。首先确定输入数字串长度的目的就是为了便于从低位向高位构造 BCD 数，从而保证整数输入将被转换为其精确的 BCD 整数值。如果输入的是一个浮点数，那么，就需要为该数字串取适当的比例因子，即根据小数点在数字串中的位置，对指数部分进行修正。

下面给出程序清单及其流程图。

```
LINE  SOURCE
   1  $title(ASCII to floating point conversion)
   2  ;
   3  ; Define the publicly known names.
   4  ;
   5          name  ascii_to_floating
   6          public ascii_to_floating
   7          extrn get_po_wer10:near
   8  ;
   9  ;       This function will convert an ASCII character string to a floating
  10  ; point representation. Character strings in integer or scientific form
```


ascii-to-floating
从字符串中取出首字符
(可能是符号+或-)
扫描小数点前面的数
字并记下数字的个数
给 BCD 设置初值
第一个非数字字符是小数点?
N
Y
扫描小数点后面的数字
碰到过数字吗?
N
Y
形成返回值NO-NUMBER
(非数值)
8087栈顶:无定义代码
数字位数太多吗?
N
Y
删去小数点后面的数尾部的0
数字串为全 0 吗?
Y
N
删去数串头上的零
位数太多吗?
Y
N
形成返回值 TOO-MANY-DIGITS
8087栈顶:无定义代码
形成 BCD 数值串
要指数吗?
N
Y
从 ASCII 字符串中取出指数值
指数太大吗?
Y
N
计算出指数值
形成实数结果
形成返回值NUMBER-FOUND
8087栈顶: 浮点数结果
形成返回值
EXPONENT-TOO-BIG
8087 栈顶: ±∞
RET

```
11  ;  will be accepted. The allowed format is.  本程序将一个 ASCII 字符串转换为浮点
       表示格式。字符串的一般格式为：
12  ;
13  ;  [+, -][digit(s)][.][digit(s)][E, e][+, -][digit(s)]  [+, -][数字…数
       字][.][数字…数字][E, e][+, -][数字…数字]
14  ;
15  ;       where a digit must have been encountered before the exponent
16  ;  indicator 'E'. or 'e'. If a '+', '-' or '.' was encountered, then at
17  ;  least one digit must exist before the optional exponent field. A value
18  ;  will always be returned in the 8087 stack. In case of invalid numbers,
19  ;  values like indefinite or infinity will be returned.  指数标记符 E 或 e 的前面必须
       有至少一个数字。返回值位于 8087 的堆栈上。
20  ;
21  ;       The first character not fitting within the format will terminate the
22  ;  conversion. The address of the terminating character will be returned
23  ;  by this subroutine.  一旦发现某个不符合规定格式的字符，就终止转换过程，并指
       出该字符的地址。
24  ;
25  ;       The result will be left on the top of the NPX stack. This
26  ;  subroutine expects 3 free NPX stack registers. The sign of the result
27  ;  will correspond to any sign characters in the ASCII string. The rounding
28  ;  mode in effect at the time the subroutine was called will be used for
29  ;  the conversion from base 10 to base 2. Up to 18 significant decimal
30  ;  digits may appear in the number. Leading zeroes, trailing zeroes, or
31  ;  exponent digits do not count towards the 18 digit maximum. Integers
32  ;  or exactly representable decimal numbers of 18 digits or less will be
33  ;  exactly converted. The technigue used constructs a BCD number
34  ;  representing the significant ASCII digits of the string with the decimal
35  ;  point removed.  结果放在 8087 的栈顶。本过程共用到 8087 的三个栈单元，结果
       符号对应于 ASCII 字符串中的符号。该数值中最多包含 18 个有效的 10 进制数位，
       这不包括打头的 0、尾上的 0、指数部分的数字位数。整数或可以用 18 位精确地表示
       出来的 10 进制数都将精确地被转换。得到的 BCD 数表示的是有效 ASCII 数字(移
       去了其中的小数点)。
36  ;
37  ;       An attempt is made to exactly convert relatively small integers or
38  ;  small fractions. For example the values,. 06125, 123456789012345678,
39  ;  le 17, 1.23456e5, and 125e-3 will be exactly converted to floating point.
40  ;       The exponentiate instruction is used to scale the generated BCD
41  ;  to very large or very small numbers. The basic accuracy of this function
```

```
42  ; determines the accuracy of this subroutine. For very large or very small
43  ; numbers, the accuracy of this function is 2 units in the 16th decimal
44  ; place or double precision. The range of decimal powers accepted is
45  ; 10**-4930 to 10**4930.  为了从产生的 BCD 数值形成很大或很小的数值，我们
      利用了指数指令.可接受的 10 进制幂值的取值范围是：10**(-4930)到 10**4930
46  ;
47  ;       The PLM/86 calling format is;
48  ;
49  ; ascii_to_floating:
50  ;       procedure(string_ptr, end_ptr, status_ptr)real external;
51  ;       declare(string_ptr, end_ptr, status_ptr)pointer;
52  ;       declare end based end_ptr pointer;
53  ;       declare status based status_ptr word;
54  ;       end;
55  ;
56  ; The status value has 6 possible states:  几种可能的状态：
57  ;
58  ;       0  A number was found.  0 发现一个数值
59  ;       1  No number was found, return indefinite.  1 没发现数值，返回无定义
60  ;       2  Exponent was expected but none found, return indefinite.  2 指数应该出
      现但没出现，返回无定义
61  ;       3  Too many digits were found, return indefinite.  3 发现数字位数太多，返
      回无定义
62  ;       4  Exponent was too big, return a signed infinity.  4 指数过大，返回一个带
      符号的无穷大
63  ;
64  ;       The following registers are used by this subroutine;
65  ;
66  ; ax  bx  cx  dx  si  di
67  ;
68
69  ;
70  ;       Define constants.  定义常数：
71  ;
72  LOW_EXPONENT        equ  -4930  ; Smallest allowed power of 10
                                      10 的最小幂值是 -4930
73  HIGH_EXPONENT       equ  4930   ; Largest allowed power of 10
                                      10 的最大幂值是 4930
74  WORD_SIZE           equ  2
```

```
75  BCD_SIZE            equ  10
76  ;
77  ;       Define the parameter layouts involved.  定义参数格式
78  ;
79  bp_save             equ  word ptr [bp]
80  return_ptr          equ  bp_save + size bp_save
81  status_ptr          equ  return_ptr + size return_ptr
82  end_ptr             equ  status_ptr + size status_ptr
83  string_ptr          equ  end_ptr + size end_ptr
84
85  parms_size          equ  size status_ptr + size end_ptr + size string_ptr
86  ;
87  ;       Define the local variable data layouts  定义局部变量数据格式;
88  ;
89  power_ten           equ  word ptr [bp_WORD_SIZE]; power of ten value
                             power_ten. 10 的幂值
90  bcd_form            equ  tbyte ptr power_ten_BCD_SIZE; BCD representat-
                             ion  bcd_form. BCD 表示
91
92  local_size          equ  size power_ten + size bcd_form
93  ;
94  ;       Define common expressions used  定义公用的表达式;
95  ;
96  bcd_byte            equ  byte ptr bcd_form      ; Current byte in the BCD
                                                      form  bcd_byte. BCD 格
                                                      式中的当前字节
97  bcd_count           equ  (type(bcd_form) - 1)*2; Number of digits in BCD
                                                      form  bcd_count.BCD 格
                                                      式中的 10 进制位数
98  bcd_sign            equ  byte ptr bcd_form + 9  ; Address of BCD sign byte
                                                      bcd_sign. BCD 符号字节
                                                      的地址
99  bcd_sign_bit        equ  80H
100 ;
101 ;       Define return values.  定义返回值;
102 ;
103 NUMBER_FOUND     equ  0                      ; Number was found 发现数
                                                   值
104 NO_NUMBER        equ  1                      ; No number was found
```

```
                                                    没发现数值
105  NO_EXPONENT            equ  2                  ; No exponent was found when
                                                    expected 指数应该出现但没
                                                    有出现
106  TOO_MANY_DIGITS        equ  3                  ; Too many digits were found
                                                    数字位数太多
107  EXPONENT_TOO_BIG       equ  4                  ; Exponent was too big 指数太
                                                    大
108  ;
109  ; Allocate stack space to insure enough exists at run time. 分配足够的栈空间
110  ;
111  stack                  segment stack 'stack'
112                         db(local_size + 4)dup(?)
113  stack                  ends
114
115  cgroup                 group   code
116  code                   segment public 'code'
117                         assume cs:cgroup
118  ;
119  ; Define some of the possible return values. 定义几个可能的返回值;
120  ;
121                         even                    ; Optimize 16 bit access
122  indefinite             dd      0FFC00000R      ; Single precision real for indefinite
                                                    代表无定义的单精度实数
123  infinity               dd      07FF80000R      ; Single precision real for + infinity
                                                    代表+∞的单精度实数
124
125  ascii_to_floating proc
126
127    fldz                                         ; Prepare to zero BCD value ST
                                                    <=0
128    push  bp                                     ; Save callers stack environment
                                                    保护调用程序的堆栈环境
129    mov   bp, sp                                 ; Establish stack addressibility 建
                                                    立堆栈的寻址机构
130    sub   sp, iocal_size                         ; Allocate space for local variables
                                                    为局部变量分配空间
131  ;
132  ; Get any leading sigh character to form initial BCD template. 取出代表符号的字
```

```
        符,形成 BCD 的初值
133     ;
134     mov     si, string_ptr          ; Get starting address of the number  取得数的始
                                          地址
135     xor     dx, dx                  ; Set initial decimal digit count  初始化 10 进制位
                                          数计数器(=0)
136     cld                             ; Set autoincrement mode  设置自增模式
137     ;
138     ; Register usage,  寄存器的用途说明
139     ;
140     ;       al:  Current character value being examined   al: 正在检查的字符值
141     ;       cx:  Digit count before the decimal point     cx: 小数点之前的位数
142     ;       dx:  Total digit count                        dx: 总位数
143     ;       si:  pointer to character string              si: 字符串指针
144     ;
145     ; Look for an initial sign and skip it if found.  找原符号,若找到符号则跳过它
146     ;
147     lodsb                           ; Get first character  从数字串中取出第一个字符
148     cmp     al, '+'                 ; Look for a sign  若为符号"+",则跳转
149     jz      scan_leading_digits
150
151     cmp     al, '-'
152     jnz     enter_leading_digits    ; If not "-" test current character  若不是符号,则
                                          转去检查当前字符
153
154     fchs                            ; Set TOS = -0  置 ST = -0
155     ;
156     ; Count the number of digits appearing before an optional decimal point.  计数小数
        点之前的数字位数
157     ;
158     scan_leading_digits:
159
160     lodsb                           ; Get next character  取数字串的下一个字符
161
162     enter_leading_digits:
163
164     call    test_digit              ; Test for digit and bump counter  是数字吗? 若是
                                          则计数器加 1
165     jnc     scan_leading_digits
```

```
166  ;
167  ;       Look for a possible decimal point and start fbstp operation
168  ; The fbstp zeroes out the BCD value and sets the correct sign  查看是否为小数点，
       并开始 fbstp 操作，fbstp 把 BCD 值置 0 且置上正确的符号
169  ;
170     fbstp   bcd_form            ; Set initial sign and value of BCD number  设置
                                      BCD 数的初始符号及初值
171     mov     cx, bx              ; Save count of digits before decimal point  保存小
                                      数点前面的数值位数
172     cmp     al, '.'
173     jnz     test_for_digits
174  ;
175  ; Count the number of digits appearing after the decimal point.  计数小数点后面出
       现的数字位数
176  ;
177  scan_trailing_digits:
178
179     lodsb                       ; Look at next character  查看下一个字符
180     call    test_digit          ; Test for digit and bump counter  看其是否为数
                                      字，若是则计数器加 1
181     jnc     scan_trailing_digits
182  ;
183  ; There must be at least one digit counted at this point  至此，可以确定是否已经碰
       到过数字
184  ;
185  test_for_digits:
186
187     dec     si                  ; Put si back on terminating character  索引 si 退
                                      回到结束字符
188     or      dx, dx              ; Test digit count  检查位数计数器的内容
189     jz      no_number_found     ; Jump if no digits were found  若为 0，表示没有
                                      发现数字
190
191     push    si                  ; Save pointer to terminator  保护终止符的指针
192     dec     si                  ; Backup pointer to last digit  把指针 si 退回到最
                                      后一个数字
193  ;
194  ;       Check that the number will fit in the 18 digit BCD format
195  ; CX becomes the initial scaling factor to account for the implied
```

```
196  ; decimal point.  检查此数是否可用 18 位的 BCD 格式表示,cx(小数点前的位数)可
       作为原始的比例因子
197  ;
198    sub    cx, dx                    ; For each digit to the right of the
199                                     ; decimal point, subtract one from the
200                                     ; initial scaling power
                                          对小数点右面的每一位,原比例因子的指数
                                          总要减 1
201    neg    dx                        ; Use negative digit count so the
202                                     ; test_digit routine can count dx up
203                                     ; to zero  计数值取反,以使 test_digit 子程
                                          序能够把 dx 加到 0
204    cmp    dx, -bcd_count            ; See if too many digits found  检查位数是
                                          否太多
205    jb     test_for_unneeded_digits
206  ;
207  ;       Setup initial register values for scanning the number right to left
208  ; while building the BCD value in memory.  为了在存贮器中构造 BCD 数时,能够
       从右向左扫描数值,故设置寄存器的一些初值
209  ;
210  form_bcd_value:
211
212    std                              ; Set autodecrement mode  设置自减模式
                                          (从右向左)
213    mov    power_ten cx              ; Set initial power of ten  设置 10 的幂的初
                                          值
214    xor    di, di                    ; Clear BCD number index  清除 BCD 数的
                                          索引
215    mov    cl, 4                     ; Set digit shift count  设置每个 10 进制数
                                          的移位次数
216    fwait                            ; Ensure BCD store is done  等待,保证存放
                                          BCD 操作的完成
217    jmp    enter_digit_loop
218  ;
219  ;       No digits were encountered before testing for the exponent.
220  ; Restore the string pointer and return an indefinite value.  在对指数部分检查之
       前,没有遇到任何数字,故恢复字符串的指针,返回一个无定义值
221  ;
222  non_umber_found:
```

```
223
224     mov     ax, NO__NUMBER          ; Set return status  设置返回值的状态
225     fld     indefinite              ; Return an indefinite numeric value  返回
                                          一个无定义的数值
226     jmp     exit
227 ;
228 ;       Test for a number of the form ????00000。检测格式为 ????00000.的数
229 ;
230 test__terminating__point:
231
232     lodsb                           ; Get last character  取出最后一个字符
233     cmp     al, '.'                 ; Look for decimal point  若为小数点,则跳转
234     jz      enter__power__zeroes    ; Skip forward if found
235
236     inc     si                      ; Else bump pointer back  否则,指针往后移
                                          动
237     jmp     short enter__power__zeroes
238 ;
239 ;       Too many decimal digits encountered. Attempt to remove leadin; and
240 ; trailing digits to bring the total into the bounds of the BCD format.  数值位数太
    多,设法去除头上及尾上的一些数位,使其可用 BCD 格式表示出来
241 ;
242 test__for__unneeded__digits:
243
244     std                             ; Set autodecrement mode  设置自减模式
245     or      cx, cx                  ; See if any digits appeared to the
246                                     ; right of the decimal point  检查小数点右
                                          面是否出现过任何数字
247     jz      test__terminating__point ; Jump if none exist  若没有则跳转
248
249     dec     dx                      ; Adjust digit counter for loop  否则,修改
                                          位数计数器的内容
250 ;
251 ;       Scan backwards from the right skipping trailing zeroes
252 ; If the end of the number is encountered, dx = 0  the string consists of
253 ; all zeroes!  从右向左扫描,去除尾巴上的所有 0
254 ;
255 skip__trailing__zeroes:
256
```

```
257     inc     dx                      ; Bump digit count  若发现数字串为全0，则跳转
258     jz      look_for_exponent       ; Jump if string of zeroes found;
259
260     lodsb                           ; Get next character  取下一字符
261     inc     cx                      ; Bump power value for each trailing  每从尾巴上
                                          去掉一位，指数值总要加1
262     cmp     al, '0'                 ; zero dropped  若字符为0，则删去
263     jz      skip_trailing_zeroes    重复上述过程
264
265     dec     cx                      ; Adjust power counter from loop  修正指数计数值
266     cmp     al, '.'                 ; Look for decimal point  字符若为小数点，则调整
                                          位数计数器
267     jnz     scan_leading_zeroes     ; Skip forward if none found  否则，跳转
268
269     dec     dx                      ; Adjust counter for the decimal point
270 ;
271 ;       The string is of the form: ????.0000000
272 ; See if any zeroes exist to the left of the decimal point.  至此，说明数字串的形式
    为：????.0000000 再检查小数点的左面是否有0存在
273 ;
724 enter_power_zeroes:
275
276     dec     dx                      ; Adjust digit counter for loop  修正位数计数器以
                                          用于循环之中
277
278 skip_power_zeroes:
279
280     inc     dx                      ; Bump digit count  位数计数值加1
281     jz      look_for_exponent
282
283     lodsb                           ; Get next character  取出下一个字符
284     inc     cx                      ; Bump power value for each trailing  每删去一位，
                                          指数计数值总要加1
285     cmp     al,'0'                  ; zero dropped  若字符为0，则删去
286     jz      skip_power_zeroes
287
288     dec     cx                      ; Adjust power counter from loop  对循环中形成
                                          的指数计数值进行修正
289 ;
```

```
290  ;       scan the leading digits from the left to see if they are zeroes.  从左面(数头)开
         始,检查它们是否为 0
291  ;
292  scan_leading_zeroes:
293
294    lea     di, byte ptr [si+1]     ; Save new end of number pointer  保护数值指针
                                         的新端点
295    cld                            ; Set autoincrement mode  设置自增模式
296    mov     si, string_ptr         ; Set pointer to the start  令指针指向起点
297    lodsb                          ; Look for sign character  检测符号字符
298    cmp     al, '+'
299    je      skip_leading_zeroes
300
301    cmp     al, '-'
302    jne     enter_leading_zeroes:
303  ;
304  ;       Drop leading zeroes. None of them affect the power value in cx.
305  ; we are guarenteed at least one non zero digit to terminate the loop.  删除数头上
         的 0,此时 cx 中的指数计数值不受影响
306  ;
307  skip_leading_zeroes:
308
309    lodsb                          ; Get next character  取下一个字符
310
311  enter_leading_zeroes:
312
313    inc     dx                     ; Bump digit count  位数计数器加 1
314    cmp     al, '0'                ; Look for a zero  若字符为 0,则删去,重复这一过
                                         程
315    jz      skip_leading_zeroes
316
317    dec     dx                     ; Adjust digit count from loop  调整位数计数器
318    cmp     al, '.'                ; Look for 000.??? form 若格式不是 000.???,则跳转
319    jnz     test_digit_count
320  ;
321  ;       Number is of the form 000. ????
322  ; Drop all leading zeroes with no effect on the power value.  否则,继续删除头上的
         0,它对指数值没有影响
323  ;
```

```
skip_middle_zeroes:

    inc     dx                          ; Remove the digit  移去当前的数据位
    lodsb                               ; Get next character  取得下一个字符
    cmp     al, '0'
    jz      skip_middle_zeroes

    dec     dx                          ; Adjust digit count from loop  调整位数
                                          计数值
;
;       All superflous zeroes are removed. Check if all is well now.  至此，所有可删
    去的 0 都已去除，再检查结果是否满足要求
;
test_digit_count:

    cmp     dx,bcd_count
    jb      too_many_digits_found

    mov     si, di                      ; Restore string pointer  恢复字符串指针
    jmp     form_bcd_value

too_many_digits_found:

    fld     indefinite                  ; Set return numeric value  设置返回值
                                          (位数太多，无定义)
    mov     ax, TOO_MANY_DIGITS         ; Set return flag  设置返回标志
    pop     si                          ; Get last address  取回上次的地址
    jmp     exit
;
;       Build BCD form of the decimal ASCII string from right to left with
; trailing zeroes and decimal point removed. Note that the only non
; digit possible is a decimal point which can be safely ignored.
; Test_digit will correctly count dx back towards zero to terminate
; the BCD build function.  从右往左构造 10 进制 ASCII 字符串的 BCD 数，非数值
    字符只可能是小数点，而小数点是可以被安全地省去的
;
get_digit_loop:

    lodsb                               ; Get next character  取得下一个字符
```

```
359     call    test__digit             ; Check if digit and bump digit count  检测是否为
                                          数字字符,且位数计数器加 1
360     jc      get__diget__loop        ; Skip the decimal point if found  跳过小数点
361
362     shl     al, cl                  ; put digit into high nibble  把这个数字送到高位
                                          部分
363     or      ah, al                  ; Form BCD byte in ah  在 ah 中形成 BCD 字节
364     mov     bcd__byte [di], ah      ; Put intoBCD string  将 BCD 字节送至 BCD 串中
365     inc     di                      ; Bump BCD pointer  BCD 串的指针增 1
366     or      dx, dx                  ; Check if digit is available  检查后面是否还有数字
367     jz      look__for__exponent
368
369 enter__digit__loop:
370
371     lodsb                           ; Get next character  取得下一个字符
372     call    test__digit             ; Check if digit  看其是否是数字
373     jc      enter digit__loop       ; skip the decimal point  跳过小数点
374
375     mov     ah, al                  ; Save digit  把这位数字保存到 ah 中,待形成 BCD
                                          字节
376     or      dx,dx                   ; Check if digit is availadle  检查后面是否还有数字
377     jnz     get__digit__loop
378
379     mov     bcd__byte[di], ah       ; Save last odd digit  将最后一个数字送到 BCD 串
                                          中
380 ;
381 ;       Look for an exponent indicator.  寻找指数标识符(E 或 e)
382 ;
383 look__for__exponent:
384
385     pop     si                      ; Restore string pointer  恢复字符串的指针
386     cld                             ; Set autoincrement direction  设置自增模式
387     mov     di,power__ten           ; Get current power of ten  取得当前的10的幂次值
388     lodsb                           ; Get next character  取下一个字符
389     cmp     al, 'e'                 ; Look for exponent indication  看其是否为指数标
                                          识符
390     je      exponent__found
391
392     cmp     al, 'E'
```

```
393     jne     convert
394     ;
395     ;       An exponent is expected, get its numeric value.  找到E或e，希望后面跟着
                指数值
396     ;
397     exponent_found:
398
399     lodsb                           ; Get next character  取得下一个字符
400     xor     di,di                   ; Clear power variable  清除幂次变量
401     mov     cx, di                  ; Clear exponent sign flag and digit flag  清除指数
                                          的符号标志及数位标志
402     cmp     al,'+'                  ; Test for positive sign  字符为正号吗?
403     je      ship_power_sign
404
405     cmp     al, '-'                 ; Test for negative sign  检测字符是否为负号
406     jne     enter_power_loop
407     ;
408     ;       The exponent is negative.  负指数
409     ;
410     inc     ch                      ; Set exponent sign flag  设置指数的符号标志(一)
411
412     skip_power_sign:
413     ;
414     ;       Register usage:  寄存器的用途说明:
415     ;
416     ;               al:  exponent character being examined  al: 正被检测的指数字符
417     ;               bx:  return value  bx: 返回值
418     ;               ch:  exponent sign flag 0 positive, 1 negative  ch: 指数符号0表示正,
                             1表示负
419     ;               cl:  digit flag 0 no digits found, 1 digits found  cl: 数字标志0表示没
                             发现数字,1反之
420     ;               dx:  not usable since test_digit increments it
421     ;               si:  string pointer  si: 字符串指针
422     ;               di:  binary value of exponent  di: 指数的2进制值
423     ;
424     ;       Scan off exponent digits until a non-digit is encountered.
425     ;
426     power_loop:
427
```

```
428     lodsb                                   ; Get next character  取下一个字符
429
430 enter_power_loop:
431
432     mov     ah,0                            ; Clear ah since ax is added to later
433     call    test_digit                      ; Test for a digit  检测它是否为数字.
                                                  字符
434     jc      form_power_value                ; Exit loop if not  若不是，跳出，转
                                                  去计算指数值
435
436     mov     cl,1                            ; Set power digit flag  是数字，则在
                                                  cl 中做上标记
437     sal     di,1                            ; old*2  计算 10 的幂次值
438     add     ax,di                           ; old*2 + digit
439     sal     di,1                            ; old*4
440     sal     di,1                            ; old*8
441     add     di,ax                           ; old*10 + digit  新幂值 = 老幂值 *10
                                                  + 当前数字
442     cmp     di,HIGH_EXPONENT + bcd_count    ; Check if exponent is too big  检查
                                                  指数是否过大
443     jna     power_loop                      不太大，继续这一循环
444 ;
445 ;       The exponent is too large.          否则，即指数太大
446 ;
447 exponent_overflow:
448
449     mov     ax, EXPONENT_TOO_BIG            ; Set return value  设置返回值
450     fld     infinity                        ; Return infinity  返回无穷大
451     test    bcd_sign,bcd_sign_bit           ; Return correctly signed infinity  返
                                                  回正确的带符号的无穷大代码
452     jz      exit                            ; Jump if not
453
454     fchs                                    ; Return-infinity
455     jmp     short exit
456 ;
457 ;       No exponent was found.  没有找到指数部分
458 ;
459 no_exponent_found:
460
```

```
461     dec     si                      ; put si back on terminating character  si 退到
                                        结束字符
462     mon     ax, NO__EXPONENT        ; Set return value  置返回值 (NO__EXPONE-
                                        NT)
463     fld     indefinite              ; Set number to return  设置返回的数值 (无定
                                        义)
464     jmp     short exit
465 ;
466 ;       The string examination is complete.Form the correct power of ten.  字符串
        的检查已告完成,形成 10 的指数值
467 ;
468 form__power__value:
469
470     dec     si                      ; Backup string pointer to terminating
471                                     ; chrracter
                                        字符串指针 si 指向终止字符
472     rcr     ch, 1                   ; Test exponent sign flag  测试指数部分的符
                                        号标记
473     jnc     positive__exponent
474
475     neg     di                      ; Force exponent negative  指数为负, 故取反
476
477 positive__exponent:
478
479     rcr     cl, 1                   ; Test exponent digit flag  检查指数的数字标
                                        记
480     jnc     no__exponent__found     ; If zero then no exponent digits were
                                        ; found
481                                     若为零,则表示没找到指数值,跳转
482     add     di, power__ten          ; Form the final power of ten value  否则, 形
                                        成最终的 10 的幂次值
483     cmp     di, LOW__EXPONENT       ; Check if the value is in range  查看该值是否
                                        在其规定范围之内
484     js      exponent__overflow      ; Jump if exponent is too small  若太小则跳转
485
486     cmp     di, HIGH__EXPONENT
487     jg      exponent__overflow  若太大,也跳转
488
489     inc     si                      ; Adjust string pointer  否则, 调整字符串的
```

指针 si

```
490  ;
491  ;       Convert the base 10 number to base 2.
492  ; Note: 10**exp = 2**(exp*log2 (10))  把以十为基化为以二为基的指数 10**exp =
       2**(exp*log2 10)
493  ;
494  ; di has binary power of ten value to scale the BCD value with.
495  ;
496  convert:
497
498    dec    si                    ; Bump string poingter back to last character  让字
                                       符串指针指向最后一个字符
499    mov    ax, di                ; Set power of ten to calculate
500    or     ax, ax                ; Test for positive or negative value  检查指数的
                                       符号
501    js     get_negative_power
502  ;
503  ;       Scale the BCD value by a value> = 1.
504  ;
505    call   get_power_10          ; Get the adjustm n power of ten  获得10的指数值
506    fbld   bcd_form              ; Get the digits to use  取待用的数字
507    fmul                         ; Form converged result  形成结合起来的结果
508    jmp    short done
509  ;
510  ;       Calculate a power of ten value >1 then divide the BCD value with
511  ; it. This technique is more exact than multiplying the BCD value by
512  ; a fraction since no negative power of ten can be exactly represented
513  ; in binary floating point. Using this technique will quarentee exact
564  ; conversion of values like. 5 and.0625。对于负指数,先求 10 的指数值,然后将它与
       BCD 数值相除,形成结果
515  ;
516  get_negative_power:
517
518    neg    ax                    ; Force positive power  负指数取反,让幂次为正
519    call   get_power_10          ; Get the adjustment power of ten  形成 10 的指数
                                       值
520    fbld   bcd_form              ; Get the digits to use  取 BCD 数字
521    fdivr                        ; Divide fractions  因为指数为负,故做除法
522    fxch                         ; Negate scale factor
```

```
523     fchs
524     fxch
525     ;
526     ;       All done, set return values.
527     ;
528     done:
529
530     fscale                          ; Update exponent of the result  修改结果的指数
531     mov     ax, NUMBER_FOUND        ; Set return value  设置返回值
532     fstp    st(1)                   ; Remove the scale factor
533
534     exit:
535
536     mov     di, status_ptr          ; Set status of the conversion  设置转换过程的
                                          标志状态
537     mov     word ptr [di], ax
538     mov     di, end_ptr             ; Set ending string address  设置字符串的结束地
                                          址
539     mov     word ptr [di], si
540     mov     sp,bp                   ; Deallocate local storage area  归还局部存贮器
                                          区域
541     pop     bp                      ; Restore caller's environment  恢复调用者的环
                                          境
542     fwait                           ; Insure all loads from memory are done  保证所
                                          有操作的完成
543     ret     parms_size
544     ;
545     ;       Test if the character in al is an ASCII digit.
546     ; If so then convert to binary, bump dx, and clear the carry flag.
547     ; Else leave as is and set the carry flag  检测 al 中的字符是否为一个 ASCII 数字。
        若是数字，则将其转换为 2 进制表示，且位数计数器加 1，并清 0 进位标志。否则设
        置进位标志
548     ;
549     test_digit:
550     cmp     rl, '9'                 ; See if a digit  检查 al 中的字符是否为数字(在
                                          '0' 到 '9' 之间)
551     ja      not_digit
552
553     cmp     al, '0'
```

```
554     jb    not__digit
555   ;
556   ;       Character is a digit.  是一个数字字符
557   ;
558     inc   dx          ; Bump digit count  位数计数器的内容增 1
559     sub   al, '0'     ; Convert to binary and clear carry flag  把该数字字符用 2 进制
                            数表示且清除进位标志
560
561   ;
562   ;       Character is not a digit.  该字符不是一个数字
563   ;
564   not__digit:
565     stc               ; Leave as is and set the carry flag  置上进位标志位
566     ret
567
568   ascii__to__floating endp
569   code    ends
570           end
ASSEMBLY COMPLETE, NO ERRORS FOUND
```

例 5：

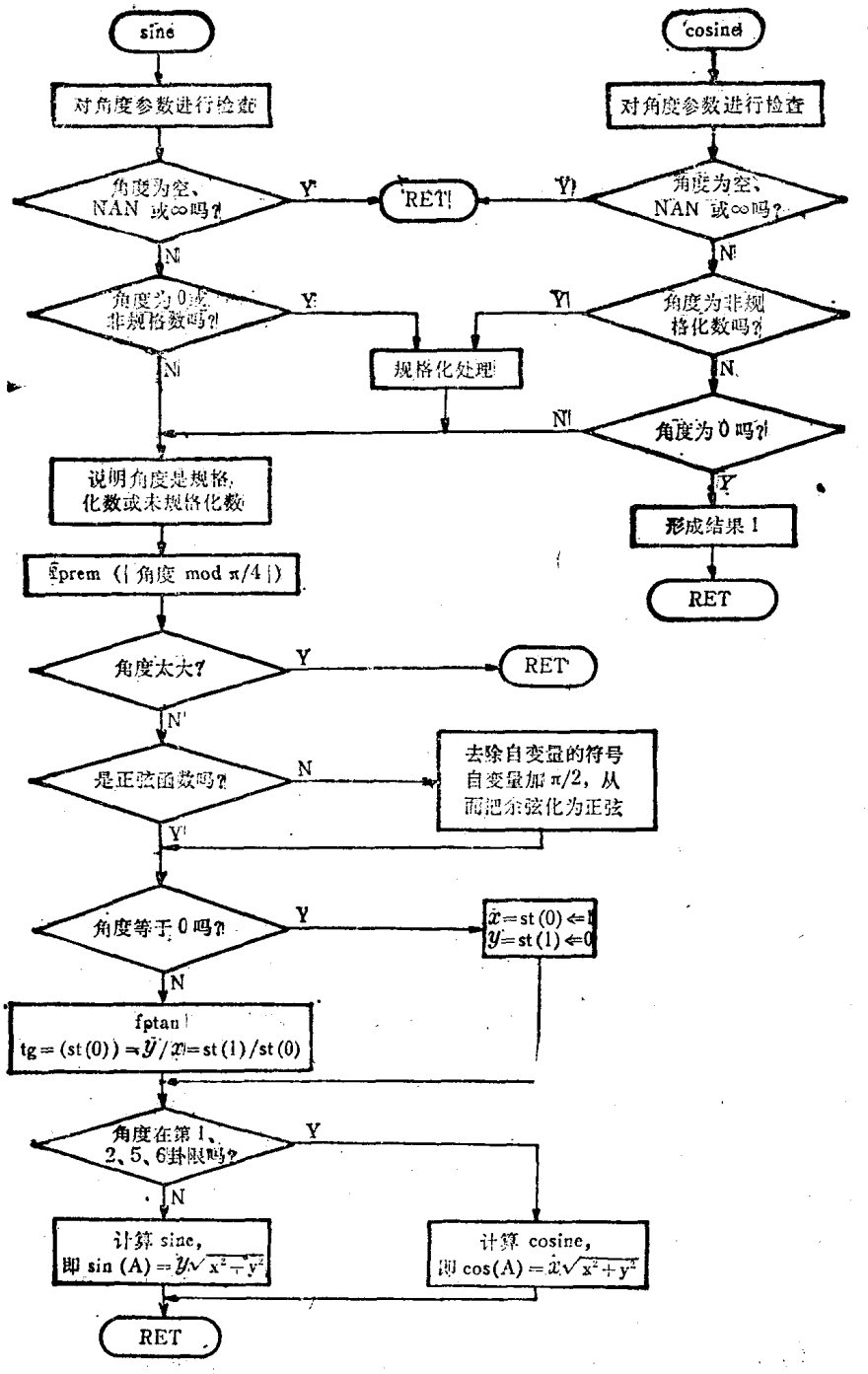
本例解决了三种三角函数值的计算问题，即正弦、正切和余弦函数值的计算．其自变量角度的取值范围为 -2^{62} 到 $+2^{62}$ 之间．这几个子程序都用到了 8087 的部分正切指令 FPTAN，当然，它们还分别用到一些三角公式．在介绍指令的时候已经讲过，指令 FPTAN 要求角度自变量在 0 到 PI/4 之间取值，即自变量的取值范围是 0 到 45°，为了达到这一要求，程序中使用了 FPREM 指令，以把角度的幅值压缩到这一范围之内，即对角度自变量用 $\pi/4$ 作模除．如果指令 FPREM 得到了一个比模数($\pi/4$)小的余数，则说明该模除操作过程结束，此时清除状态字中的条件码 C2 位；若此取模操作是不完全的，即结果仍然大于模数($\pi/4$)，那么，状态字中的 C2 位就被置 1．取模操作得到的部分余数放在 8087 栈顶寄存器当中．此外，FPREM 指令还提供了模除产生的商的三个最低有效位(C_3、C_1、C_0)，这对本例程也是很有用处的，它有助于正确地把原角度自变量定位在某个卦限中(单位圆的八个 $\pi/4$ 段之一中)．

为了节省代码空间，函数 cosine(余弦)使用了函数 sine(正弦)的大部分代码，而这样做的根据就是：利用关系 sin (|A| + PI/2) = cos(A)就可以把余弦函数化为正弦函数．本例中角度加 PI/2 过程的实现方法是，将 010_2 加到 FPREM 形成的三个最低商位上．

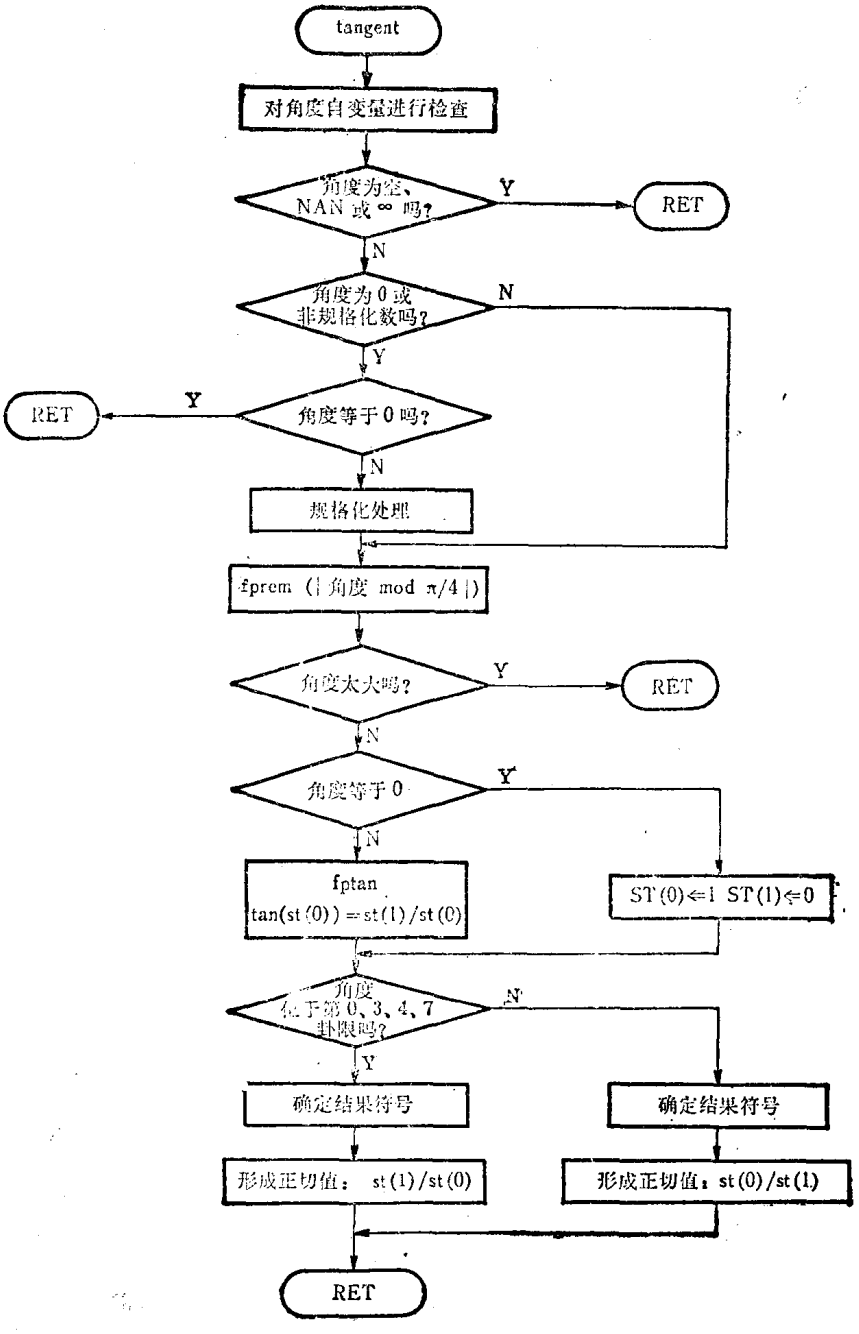
在下面几个函数当中，都比较多地用到主机与 8087 之间的并发执行特色，即在 8087 执行数值运算操作的同时，主机有效地完成一些控制方面的工作，正如前面曾经指出的那样，这也就产生了同步的维持问题．

下面给出这几个三角函数的处理流程图及其汇编源程序代码．

```
LINE      SOURCE
   1      Stitle(8087 Trignometric Functions)
```




```
2
3               public sine, cosine, tangent
4               name trig_functions
5
6   +1 sinclude(,fl,8087.anc)
7       ;
8       ;       Define 8087 word packing in the environment area.  定义装在环境区域中的
```



8087 寄存器的格式

```
9       ;8087 的控制字格式
10      cw_87 record res87: 13, infinity_control: 1, rounding_control: 2,
11      &             precision_control: 2, error_enable: 1, res872: 1,
12      &             precision_mask: 1, underflow_mask: 1, overflow_mask: 1,
13      &             zero_divide_mask: 1, denormal_mask: 1, invalid_mask: 1
```

```
14                          8087 的状态字格式
15  sw_87                   record busy: 1, cond3: 1, top: 3, cond2: 1, cond1: 1, cond0: 1,
16  &                               error_pending:1, res873:1, precision_error:1,
17  &                               underflow_error: 1, overflow_error: 1, zero_divide_
                                    error:1,
18  &                               denormal_error:1, invalid_error:1
19                          8087 的标志字格式
20  tw_87                   record reg7_tag: 2, reg6_tag: 2, reg5_tag: 2, reg4_tag:2,
21  &                               reg3_tag:2, reg2_tag:2, reg1_tag:2, reg0_tag:2
22
23  low_ip_87               record low_ip: 16 事故指示器
24
25  high_ip_op_87           record hi_ip:4, res874:1, opcode_87:11
26
27  low_op_87               record low_op:16
28
29  high_op_87              record hi_op:4, res875:12
30
31  environment_87          struc    ; 8087 environemnt layout   8087 环境的格式
32  env87_cw                dw   ?
33  env87_sw                dw   ?
34  env87_tw                dw   ?
35  env87_low_ip            dw   ?
36  env87_hip_op            dw   ?
37  env87_low_op            dw   ?
38  env87_hop               dw   ?
39  environment_87          ends
40  ;
41  ;       Define 8087 related constants.  定义与 8087 有关的一些常量
42  ;
43  TOP_VALUE_INC   equ     sw_87<0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0>
44
45  VALID_TAG               equ   0 ; Tag register values
46  ZERO_TAG                equ   1
47  SPECIAL_TAG             equ   2
48  EMPTY_TAG               equ   3
49  REGISTER_MASK           equ   7
```

```
50
51  ;
52  ;       Define local variable areas.  定义局部变量区域
53  ;
54  stack           segment stack 'stack'
55
56  local_area      struc
57  sw1             dw      ?                       ; 8087 status value  8087 的状
                                                      态值
58  local_area      ends
59
60                  db      size local_area + 4     ; Allocate stack space  分配堆栈
                                                      空间
61  stack           ends
62
63  code            segment public 'code'
64                  assume  cs: code, ss: stack
65  ;
66  ;       Define local constants.  定义局部常量
67  ;
68  status          equ     [bp].sw1                ; 8087 status value location
                                                      8087 状态值的地址
69
70                  even
71
72  p_quarter       dt      3FFEC90FDAA22168C235R   ; PI/4  π/4
73  indefinite      dd      0FFC00000R              ; Indefinite special value  特殊
                                                      值"无定义"
74
75  ;
76  ;       This subroutine calculates the sine or cosine of the angle, given in
77  ; radians. The angle is in ST(0), the returned value will be in ST(0).
78  ; The result is accurate to within 7 units of the least significant three
79  ; bits of the NPX extended real format. The PLM/86 definition is;  该子程序计算正弦
      或余弦函数值,自变量以弧度为单位,角度放在 ST(0)中,返回结果也将放到 8087 堆栈
      顶 ST(0)上。它需要三个 8087 堆栈单元。
80  ;
81  ; sine;    procedure(angle)real external;  过程 sine 接收一个实数表示的角度,得到一
                个实数结果 sin(angle)
```

```
82  ;           declare angle real;
83  ;           end sine;
84  ;
85  ; cosine:  procedure (angle) real external;  过程 cosine 也有一个实数自变量（角度），
                它也将形成一个实数结果 cos(angle)
86  ;           declare angle real;
87  ;           end cosine;
88  ;
89  ;       Three stack registers are reguired. The result of the function is
90  ; defined as follows for the following arguments.  函数的结果定义如下：
91  ;
92  ;              angle （角度）                          result （结果）
93  ;
94  ;           valid or unnormal less than 2**62 in magnitude correct value  小于 2**62
                的有效值或非规格化数                        正确的结果值
95  ;           zero                                      0 或 1
96  ;           denormal                                  correct denormal
                未规格化数                                正确的未规格化值
97  ;           valid or unnormal greater than 2**62      indefinite
                大于 2**62 的有效或非规格化数               无定义
98  ;           infinity                                  indefinite
99  ;           NAN                                       NAN
100 ;           empty                                     empty
101
102 ;
103 ;       This function is based on the NPX fptan instruction. The fptan
104 ; instruction will only work with an angle of from 0 to PI/4.with this
105 ; instruction, the sine or cosine of angles from 0 to PI/4 can be accurately
106 ; calculated. The technique used by this routine can calculate a general
107 ; sine or cosine by using one of four possible operations.  本函数是以 8087 的 FPTAN
    指令为基础的，通过使用下面四种可能的操作，就能够计算出一般的正弦或余弦函数
    值：
108 ;
109 ; Let R = |angle mod PI/4|  令 R = |angle mod PI/4|
110 ;     S = -1 or 1, according to the sign of the angle  S = -1 或 1，这要由 angle 的符
                                                          号决定
111 ;
112 ; 1) sin(R)     2) cos(R)     3) sin(PI/4-R)     4) cos(PI/4-R)
113 ;
```

```
114  ;       The choice of the relation and the sign of the result follows the
115  ; decision table shown below based on the octant the angle falls in.  根据角度 (angle)
       位于哪个卦限,由下表确定应该选择哪个关系以及结果的符号。
116  ;
117  ;          octant     sine      cosine
118  ;
119  ;            0        S*1         2
120  ;            1        S*4         3
121  ;            2        S*2       -1*1
122  ;            3        S*3       -1*4
123  ;            4       -S*1       -1*2
124  ;            5       -S*4       -1*3
125  ;            6       -S*2         1
126  ;            7       -S*3         4
127  ;
128
129  ;
130  ;       Angle to sine function is a zero or unnormal.  给函数 sine 的角度为 0 或非规
       格化数
131  ;
132  sine_zero_unnormal:
133
134    fstp   st(1)                    ; Remove PI/4  移出 π/4
135    jnz    enter_sine_normalize;  Jump if angle is unnormal  若 angle (角度)是一非
                                        规格数则跳转
136  ;
137  ;       Angle is a zero.  否则,说明角度为 0
138  ;
139    pop    bp                       ; Return  the zero as the result  把 0 作为结果返回
140    ret
141  ;
142  ;       Angle is an unnormal.  角度是一个非规格化数
143  ;
144  enter_sine_normalize:
145
146    call   normalize_value
147    jmp    short enter_sine
148
149  cosine proc                       ; Entry point to cosine  函数 cosine 的入口
```

卦限　sin　cos

0　S*1　2

1　S*4　3

⋮　⋮　⋮

7　-S*3　4

↳表示结果为负,应选第三个关系,即 sin(PI/4-R); PI, π

```
150
151     fxam                        ; Look at the value  检测栈顶ST的数值
152     push    bp                  ; Establish stack addressibility  建立堆栈的寻址机构
153     sub     sp,size local_area  ; Allocate stack space for status  为状态分配堆栈空间
154     mov     bp, sp
155     fstsw   status              ; Store status value  保存状态值
156     fld     pi_quarter          ; Setup for angle reduce  设定状态减量
157     mov     cl, 1               ; Signal cosine function  标记为cosine函数
158     pop     ax                  ; Get status value  取状态值
159     lahf                        ; ZF = C3, PF = C2, CF = C0
160     jc      funny_parameter     ; Jump if parameter is  若参数为空、NAN或∞则跳转
161                                 ; empty, NAN, or infinity
162 ;
163 ;       Angle is unnormal, normal, zero, denormal.  否则，说明角度为非规格化、规格化、零、未规格化数
164 ;
165     fxch                        ; st(0) = angle, st(1) = PI/4
166     jpe     enter_sine          ; Jump if normal or denormal  若角度为规格化或未规格化的，则跳转
167 ;
168 ;       Angle is an unnormal or zero.  否则，它是非规格化数值或0
169 ;
170     fstp    st(1)               ; Remove PI/4  去除原st(1)中的π/4
171     jnz     enter_sine_normalize  若为非规格化的角度，则跳转
172 ;
173 ;       Angle is a zero. cos(0) = 1.0  否则，说明角度为0，而cos(0) = 1.0
174 ;
175     fstp    st(0)               ; Remove 0  弹出角度0
176     pop     bp                  ; Restore stack  恢复堆栈
177     fld1                        ; Return 1  返回结果1
178     ret
179 ;
180 ;       All work is done as a sine function. By adding PI/2 to the angle
181 ; a cosine is converted to a sine. of course the angle addition is not
182 ; done to the argument but rather to the program logic control values.  以下工作都是由函数sine完成的，由三角公式知：把角度加上π/2，原cosine函数就变为sine了。
183 ;
184 sine:                           ; Entry point for sine function  函数sine的入口
185
```

```
186     fxam                        ; Look at the parameter  检测参数值
187     push    bp                  ; Establish stack addressibility  建立堆栈的寻址机构
188     sub     sp,size local_area  ; Allocate local space  分配局部空间
189     mov     bp, sp
190     fstsw   status              ; Look at fxam status  保存 fxam 形成的 8087 状态值
191     fld     pi_quarter          ; Get PI/4 value  取得 PI/4
192     pop     ax                  ; Get fxam status  取得 fxam 所产生的状态
193     lahf                        ; CF = C0, PF = C2, ZF = C3
194     jc      funny_parameter     ; Jump if empty, NAN, or infinity
                                    若角度为空、NAN 或 ∞，则跳转
195 ;
196 ;       Angle is unnormal, normal, zero, or denormal.
        否则，说明角度为非规格格化化、规、零或未规格化数
197 ;
198     fxch                        ; ST(1) = PI/4, st(0) = angle
199     mov     cl, 0               ; Signal sine  标记为函数 sine
200     jpo     sine_zero_unnormal  ; Jump if zero or unnormal
                                    若角度为 0 或非规格化数，则跳转
201 ;
202 ;       ST(0) is either a normal or denormal value. Both will work.
203 ; Use the fprem instruction to accurately reduce the range of the given
204 ; angle to within 0 and PI/4 in magnitude. If fprem cannot reduce the
205 ; angle in one shot, the angle is too big to be meaningful, >2**62
206 ; radians. Any roundoff error in the calculation of the angle given
207 ; could completely change the result of this function. It is safest to
208 ; call this very rare case an error.  否则，即 ST(0)中为规格化或未规格化数值。借助
        于指令 FPREM 就可以把给出的角度调整到 0 到 π/4 的范围之内。
209 ;
210 enter_sine:
211
212     fprem                       ; Reduce angle  减小角度值 |angle mod PI/4|
213                                 ; Note that fprem will force a
214                                 ; denormal to a very small unnormal
                                    指令 fprem 将把未规格化数化为一极小的非规格化数值
215                                 ; Fptan of a very small unnormal
216                                 ; will be the same very small
217                                 ; unnormal, which is correct.
218     mov     sp, bp              ; Allocate stack space for status为保存状态分配栈空间
219     fstsw   status              ; Check if reduction was complete  检查减法是否完成
```

```
220                                     ; Quotient in C0, C3,C 1
                                          模除中商的最低三位在放 C0, C3, C1 中，以确定
                                          卦限
221     pop     bx                      ; Get fprem status  取出 fprem 的状态
222     test    bh, high(mask cond2)    ; sin(2*N*PI + X) = sin(x)
223     jnz     angle_too_big             若 C2≠0，说明角度太大，跳转
224 ;
225 ;       Set sign flags and test for which eighth of the revolution the
226 ; angle fell into.  确定符号标志及角度所位于的卦限
227 ;
228 ; Assert, -PI/4 < st(0) < PI/4   断言：  -PI/4 < st(0) < PI/4
229 ;
230     fabs                            ; Force the argument positive
                                          迫使自变量为正，符号在 bx 的 C1 中
231                                     ; cond1 bit in bx holds the sign
232     or      cl, cl                  ; Test for sine or cosine function
233     jz      sine_select             ; Jump if sine function  若为 sine 函数，则跳转
234 ;
235 ;       This is a cosine function. Ignore the original sign of the angle
236 ; and add a quarter revolution to the octant id from the fprem instruction.
237 ; cos(A) = sin(A + PI/2)and cos(|A|) = cos(A)
    否则，说明是 cosine 函数，忽略角度的符号 (因为 cos(|A|) = cos(A))，且加 PI/2 到
    指令 fprem 形成的顶上去 (因为 cos(A) = sin(A + PI/2)
238 ;
239     and     ah, not high(mask cond1); Turn off sign of argument  去除自变量的符号
240     or      bh, high(mask busy)     ; Prepare to add 010 to C0, C3, C1  为把 010 加到
                                          ax 中的 C0, C3, C1上做准备
241                                     ; status value in ax
242                                     ; Set busy bit so carry out from
243     add     bh, high(mask cond3)    ; C3 will go into the carry flag  C3 送到进位位
244     mov     al, 0                   ; Extract carry flag  抽出进位标志
245     rcl     al, 1                   ; Put carry flag in low bit  进位位送到低位
246     xor     bh, al                  ; Add carry to C) not changing
247                                     ; C1 flag
248 ;                                     把进位位加到 C0 上，不改变 C1 的状态
249 ;       See if the argument should be reversed, depending on the octant in
250 ; which the argument fell during fprem.  根据角度落在哪一卦限，确定是否需要将自
    变量反向
251 ;
```

```
sine_select:

    test    bh, high(mask cond1)        ; Reverse angle if C1 = 1  若 C1 = 1,则翻转角度
    jz      no_sine_reverse
;
;       Angle was in octants 1, 3, 5, 7  角度位于 1、3、5、7 卦限
;
    fsub                                ; Invert sense of rotation  反向旋转角度
    jmp     short do_sine_fptan         ; 0 <arg< = PI/4
;
;       Angle was in octants 0, 2, 4, 6
; Test for a zero argument since fptan will not work if st(0) = 0  若角度位于 2、4、6、
  8 卦限,先要检查自变量是否为零,这是因为若 st(0) = 0, fptan 就不能工作
;
no_sine_reverse:

    ftst                                ; Test for zero angle  检测角度是否为零
    mov     sp, bp                      ; Allocate stack space  分配栈空间
    fstsw   status                      ; cond3 = 1 if st(0) = 0  若 st(0) = 0,则 C3 = 1
    fstp    st(1)                       ; Remove PI/4  去除栈中的 PI/4
    pop     cx                          ; Get ftst status  取出 ftst 产生的状态字
    test    ch, high(mask cond3)        ; If C3 = 1, argument is zero  若 C3 = 1, 则自
                                          变量(角度)为 0, 跳转
    jnz     sine_argument_zero
;
;       Assert :0 < st(0) < = PI/4      否则,定有 0 < st(0) < = PI/4
;
do_sine_fptan:

    fptan                               ; TAN ST(0) = ST(1)/ST(0) = Y/X  Y/X = ST
                                          (1)/ST(0) = tan(ST(0))

after_sine_fptan:

    pop     bp                          ; Restore stack  恢复存贮器堆栈
    test    bh,high(mask cond3+mask cond1); Look at octant angle fell into  检查角度
                                            落入哪个卦限
    jpo     x_numerator                 ; Calculate cosine for octants  对于第 1、2、5、6
                                          卦限的角度,计算余弦值
```

```
286                                     ; 1, 2, 5, 6
287 ;
288 ;       Calculate the sine of the argument.  否则,求正弦函数值. 注意
289 ; sin(A) = tan(A)/sqrt(1 + tan(A)**2)     if tan(A) = Y/X then  sin (A) = tan(A)/
    √(1 + tan²(A)),
290 ; sin(A) = Y/sqrt(X*X + Y*Y)
291 ; 若 tan(A) = Y/X, 则有 sin(A) = Y/sqrt(X*X + Y*Y)
292   fld     st(1)                 ; Copy Y value  复写Y到 8087 栈顶
293   jmp     short finish_sine     ; Put Y value in numerator  Y作分子
294 ;
295 ;       The top of the stack is either NAN, infinity, or empty.  8087 栈顶的内容为
    NAN、∞ 或空
296 ;
297 funny_parameter:
298
299   fstp    st(0)                 ; Remove PI/4  弹出 PI/4
300   jz      return_empty          ; Return empty if no parm  若没参数(8087 栈顶为空),
                                      则返回空
301
302   jpo     return_NAN            ; Jump if st(0) is NAN  若 st(0) 为 NAN,则跳转
303 ;
304 ;       st(0) is infinity. Return an indefinite value.  否则, st(0)中为∞, 返回一个无
    定义值
305 ;
306   fprem                         ; ST (1) can be any thing
307
308 return_NAN:
309 return_empty:
310
311   pop     bp                    ; Restore stack  恢复堆栈
312   ret                           ; Ok to leave fprem running  返回, 让 fprem 继续运
                                      行
313 ;
314 ;       Simulate fptan with st(0) = 0  自变量等于 0,故其正切值也应等于 0
315 ;
316 sine_argument_zero:
317
318   fldl                          ; Simulate tan(0)  模拟执行 tan(0),即让 ST(0) = X = 1
                                      而 ST(1) = Y = 0
```

```
319     jmp    after_sine_fptan  ; Return the zero value  相当于 tan(0)的结果 Y/X = 0/
                                   1 = 0
320   ;
321   ;       The angle was too large. Remove the modulus and dividend from the
322   ; stack and return an indefinite result.  角度过大,返回一个无定义的结果
323   ;
324   angle_too_big:
325
326     fcompp                  ; pop two values from the sfack  弹出 8087 堆栈的两个
                                  单元
327     fld    indefinite       ; Return indefinite  返回值无定义
328     pop    bp               ; Restore stack  恢复存贮器堆栈
329     fwait                   ; wait for load to finish  等待 fld 取数操作的完成
330     ret
331   ;
332   ;       Calculate the cosine of the argument.  计算自变量的余弦值,注意
333   ; cos(A) = 1/sqrt(1 + tan(A)**2)  if tan(A) = Y/X then  cos(A) = 1/sqrt(1 + tan(A)
      **2),
334   ; cos(A) = X/sqrt(X*X + Y*Y)
335   ; 若 tan(A) = Y/X  则有 cos(A) = X/sqrt(X*X + Y*Y)
336   X_numerator:
337
338     fld    st(0)            ; Copy X value  把 X 复写到 8087 的栈顶
339     fxch   st(2)            ; Put X in numerator  让 X 做分子
340
341   finish_sine:
342
343     fmul   st, st(0)        ; Form X*X + Y*Y  求 X*X + Y*Y
344     fxch
345     fmul   st, st(0)
346     fadd                    ; st(0) = X*X + Y*Y  st(0) = X*X + Y*Y
347     fsqrt                   ; st(0) = sqrt(X*X + Y*Y)  st(0) = √(X*X + Y*Y)
348
349   ;
350   ;       Form the sign of the result. The two conditions are the C1 flag from
351   ; FXAM in bh and the C0 flag from fprem in ah.  形成结果的符号,它由两个条件决
      定.
352   ;
353     and    bh,high(mask cond0); Look at the fprem C0 flag  看 fprem 产生的 C0 标志
```

```
354     and     ah, high (mask condl) ; Look at the fxam C1 flag  看 fxam 产生的 C1 标志
355     or      bh, ah;               ; Even number of flags cancel  符号相同,结果为正
356     jpe     positive_sine         ; Two negatives make a positive
357
358     fchs                          ; Force result negative  否则,结果为负
359
360 positive_sine;
361
362     fdiv                          ; Form final result  形成最终结果
363     ret                           ; OK to leave fdiv running
364
365 cosine endp
366
367 ;
368 ;       This function will calculate the tangent of an angle.
369 ; The angle, in radians is passed in ST(0), the tangent is returned
370 ; in ST(0). The tangent is calculated to an accuracy of 4 units in the
371 ; least three significant bits of an extended real format number. The
372 ; PLM/86 calling format is;  下面这个函数计算某个角度的正切值。角度位于 ST(0)
    中,它以弧度为单位,正切值也将在栈顶上返回。它用到两个堆栈寄存器
373 ;
374 ; tangent; procedure(angle) real external;
375 ;           declare angle real;
376 ;           end tangent;
377 ;
378 ;       Two stack registers are used. The result of the tangent function is
379 ; defined for the following cases;  正切函数的结果定义如下;
380 ;
381 ;                       angle (角度)                result (结果)
382 ;
383 ;           valid or unnormal<2**62 in magnitude correct value
384 ;           0
385 ;           denormal                                correct denormal
386 ;           valid or unnormal>2**62 in magnitude indefinite
387 ;           NAN                                     NAN  NAN    NAN
388 ;           infinity                                indefinite 无穷大    无定义
```

```
;          empty                                   empty
;
;       The tangent instruction uses the fptan instruction. Four possible
; relations are used;        程序中用到指令 fptan,还用到下面四种可能的关系:
;
; Let R = |angle MOD PI/4|   令 R = |angle MOD PI/4|
;       S = -1 or 1 depending on the sign of the angle   S = -1 或 1, 这要由角度的符号
  决定
;
; 1) tan(R)     2) tan(PI/4-R)     3) 1/tan(R)     4) 1/tan(PI/4-R)
;
;       The following table is used to decide which relation to use depending
; on in which octant the angle fell.
  根据角度处于哪个卦限,由下表确定使用哪个关系
;
; octant  relation
;
;   0       S*1
;   1       S*4
;   2      -S*3
;   3      -S*2
;   4       S*1
;   5       S*4
;   6      -S*3
;   7      -S*2
;
tangent proc

    fxam                        ; Look at the parameter   检测参数值
    push   bp                   ; Establish stack addressibility   建立堆栈寻址机构
    sub    sp, size local_area; Allocate local variable space   分配局部变量空间
    mov    bp, sp
    fstsw  status               ; Get fxam status   保存 fxam 形成的状态
    fld    pi_quarter           ; Get PI/4
    pop    ax
    lahf                        ; CF = C0, PF = C2, ZF = C3
    jc     funny_parameter
;
;       Angle is unnormal, normal, zero, or denormal.   若角度为空、NAN 或 ∞ 则跳转
```

```
        否则，说明角度为非规格化、规格化、零、未规格化的数
426     ;
427       fxch                          ; st(0) = angle, st(1) = PI/4
428       jpe     tan__zero__unnormal
429     ; 若角度为 0 或非规格化的，则跳转。否则，角度为规格化或未规格化数。把角度减小
          到 - PI/4 到 PI/4 的范围内
430     ;       Angle is either an normal or denormal.
431     ; Reduce the angle to the range - PI/4 < result < PI/4.
432     ; If fprem cannot perform this operation in one try, the magnitude of the
433     ; angle must be > 2**62. Such an angle is so large that any rounding
434     ; errors could make a very large difference in the reduced angle.
435     ; It is safest to call this very rare case an error.
436     ;
437     tan__normal:
438
439       fprem                         ; Quotient in C0, C3, C1  商的最低位放在C0,C3,C1中
440                                     ; Convert denormals into unnormals  且把未规格化数
                                          化为非规格化数
441       mov   sp, bp                  ; Allocate stack space  分配存贮器堆栈空间
442       fstsw status                  ; Quotient identifies octant  状态字中的条件位确定了
                                          角度落入的卦限
443                                     ; original angle fell into
444       pop   bx                      ; tan(PI*N + X) = tan(X)
445       test  bh,high(mask cond2);  T st for complete reduction  看 fprem 的重复减过程是
                                          否完成
446       jnz   angle__too__big         ; Exit if angle was too big  若没完成，表示角度太大，跳
                                          转返回
447     ;
448     ;       See if the angle must be reversed.  检查角度是否必须反向
449     ;
450     ; Assert: - PI/4 < st(0) < PI/4  断言：- PI/4 < st(0) < PI/4
451     ;
452       fabs                          ; 0 < = st(0) < PI/4
453                                     ; C1 in bx has the sign flag
                                          令角度为正，其真实符号由 bx 中的 C1 标志
454       test  bh,high(mask condl);  must be reversed
455       jz    no__tan__reverse
456     ;
457     ;       Angle fell in octants 1, 3, 5, 7. Reverse it, subtract it from PI/4.  角度落在第
          1, 3, 5, 7 卦限，则从 PI/4 中减去它(反向)
```

```
458  ;
459    fsub                    ; Reverse angle  使角反转
460    jmp   short do_tangent
461  ;
462  ;      Angle is either zero or an unnormal.  给出的角度为 0 或非规格化数
463  ;
464  tan_zero_unnormal:
465
466    fstp   st(1)            ; Remove PI/4  从寄存器堆栈中去掉 PI/4
467    jz     tan_angle_zero
468  ;
469  ;      Angle is an unnormal.  角度是一非规格化数
470  ;
471    call   normalize_value
472    jmp    tan_normal
473
474  tan_angle_zero:
475
476    pop    bp               ; Restore stack  恢复存贮器堆栈
477    ret
478  ;
479  ;      Angle fell in octants 0, 2, 4, 6. Test for st(0) = 0, fptan won't work.  角度落
       在第 0、2、4、6 卦限,需检查 ST(0) 是否为 0
480  ;
481  no_tan_reverse:
482
483    ftst                    ; Test for zero angle  检测角度零
484    mov   sp, bp            ; Allocate stack spcae  分配栈空间
485    fstsw  status           ; C3 = 1 if st(0) = 0  若 st(0) = 0, 则有 C3 = 1
486    fstp   st(1)            ; Remove PI/4  移出 PI/4
487    pop    cx               ; Get ftst status  取得 ftst 建立起来的状态
488    test   ch,high(mask cond 3)
489    jnz    tan_zero
490
491  do_tangent:
492
493    fptan                   ; tan ST(0) = ST(1)/ST(0)
494
495  after_tangent:
```

```
;
;       Decide on the order of the operands and their sign for the divide
; operation while the fptan instruction is working.  在执行 fptan 指令的同时，确定除法操作中操作数的顺序和符号

    pop     bp                              ; Restore stack  恢复堆栈
    mov     al, bh                          ; Get a copy of fprem C3 flag  通过条件码确定角度处于哪一卦限，进而确定除法的方向。
    and     ax, mask cond1+high(mask cond3) ; Examine fprem C3 flag and
                                            ; fxtract C1 flag
    test    bh, high(mask cond1+mask cond3) ; Use reverse divide if in
                                            ; octants 1, 2, 5, 6
    jpo     reverse__divide                 ; Note! parity works on low
                                            ; 8 bits only!
;
;       Angle was in octants 0, 3, 4, 7.
; Test for the sign of the result. Two negatives cancel.  若角度落在第 0、3、4、7 卦限，则做普通除法得到正切值，即 ST(1)/ST(0)
;
    or      al, ah
    jpe     positive__divide  符号为正

    fchs                                    ; Force result negative  结果符号为负

positive__divide:

    fdiv                                    ; Form result  形成结果——正切值
    ret                                     ; OK to leave fdiv running

tan__zero:

    fldl                                    ; Force 1/0 = tan(PI/2)
    jmp     after__tangent
;
;       Angle was in octants 1, 2, 5, 6.
; Set the correct sign of the result.  角度若落在第 1、2、5、6 卦限，则做反向除法，即 ST(0)/ST(1)得到正切值，确定结果的符号
;
```

```
530  reverse__divide;
531
532    cr        al, ah
533    jpe       positive__r__divide   结果为正
534
535    fchs                  ; Force result negative   结果为负
536
537  positive__r__divide;
538
539    fdivr                 ; Form reciprocal of result   形成结果
540    ret                   ; OK to leave fdiv running
541
542  tangent endp
543  ;
544  ;       This function will normalize the value in st(0).
545  ; Then PI/4 is placed into st(1). 下面这段程序将规格化 st(0)中的值,然后把 PI/4 放
       到 st(1)中
546  ;
547  normalize__value;
548
549    fabs                  ; Force value positive   使 st(0)中值为正
550    fxtract               ; 0< =st(0)<1   新栈顶为原 st(0)的有效字段
551    fldl                  ; Get normalize bit
552    fadd      st(1),st    ; Normalize fraction   规格化了小数部分
553    fsub                  ; Restore original value   恢复原值
554    fscale                ; Form original normalized value   形成了规格化的原始值
555    fstp      st(1)       ; Remove scale factor   从栈中移出比例因子(原指数)
556    fld       pi__quarter; Get PI/4   取 PI/4 到 8087 堆栈上
557    fxch
558    ret
559
560  code      ends
561            end
ASSEMBLY COMPLETE, NO ERRORS FOUND
```

思　考　题

10.1　通过本章的学习,你认为 8087 在哪些应用当中可以发挥较大的作用?

10.2　8087 是怎样与 CPU 一道取出指令的? 8087 为什么设置指令流队列? 其指令队列的长度又是如何确

定的？

10.3 8087提供了哪些寄存器？其通用寄存器的访问有些什么特色？

10.4 8087提供了哪几种数据类型，各种数据类型的主要作用是什么？它们在8087内部是如何被表示的？

10.5 写一段程序，要求计算：

$$Z=\sqrt{x^2+y^2}+\frac{x(x^2+y^2)}{x+y}$$

10.6 简述8086/8088协处理器接口的构成及作用，主机与协处理器之间的关系是怎样的？

10.7 8087的总线状态信号($\overline{S2}$、$\overline{S1}$、$\overline{S0}$)和队列状态信号(QS_1、QS_0)起什么作用？

10.8 假如你想设计一个你所需要的协处理器，试叙述其设计思想，在设计过程中应该注意哪些问题？

10.9 在iAPX 86/22、88/22系统中，应如何将各处理器的RQ/GT信号线连接起来？即应考虑哪些因素？并说出理由。

10.10 对8087的事故有几种处理方法？简述8087的中断请求过程。8087的中断与一般的中断相比有些什么特点？8087的中断机构应该满足哪些条件？如何满足这些条件？

10.11 解释数值处理机中“同步”的含义，为什么要求保证指令同步、数据同步和错误同步？如何保证获得“同步”？

10 12 试编写一段程序，要求计算余弦函数的值(参见§10.5.4中的例5)。

第十一章 iAPX 86/30、iAPX 88/30 操作系统处理机

§11.1 引 言

随着LSI、VLSI及微型计算机技术的快速发展，微处理机的性能不断提高，存贮空间愈来愈大。越来越多的实践证明，对当今的绝大多数单任务、单设备应用系统来讲，利用微型计算机就足以满足其实用需要而有余。特别是在16位微型机出现之后，这一富余问题就显得更为突出。为了解决这一问题，人们很自然地想到把微处理机用于实时多任务、多设备的控制，以便尽可能充分地利用微机系统的性能，这种发展趋势向系统设计者们提出了新的挑战，即实时多道系统的开发及软件研制。经过设计者及用户的共同努力，在这方面已经取得了一些成功经验，例如，多功能的台式及个人计算机、控制电话交换线路的PABX设备、在多磁盘/多磁盘用户的系统中起协调和控制作用的用户磁盘文件子系统、以及类似于对银行存款进行管理的事务处理系统，在以上列出的这样一些系统当中，都存在着一个控制、协调整个系统工作的问题，为了解决好这一问题，就需要提供一个对多任务环境进行管理的操作系统。Intel 80130正是为了给iAPX86/10、88/10提供操作系统而专门设计的一个芯片，我们也把80130称为操作系统固件(OSF)。

iAPX 86/10加上一片操作系统固件80130就组成了操作系统处理机(OSP) iAPX 86/30，与此类似，iAPX 88/10加上一片OSF就形成了操作系统处理机iAPX 88/30。由此形成的操作系统处理机为实时多任务应用提供了一个良好的基础，也就是说，在这种系统当中，可以比较方便地实现多道程序及多任务的运行，因为操作系统处理机具有管理并发执行的多任务的能力，并为多道任务的并发执行提供了系统支持。iAPX 86/30和iAPX 88/30除了具有iAPX 86/10、88/10的所有功能之外，还提供了一个实时操作系统的核心功能，80130在CPU的基本数据类型(整型、字节、字符等)的基础上，又加上了几种新的系统数据类型(作业、任务、信箱区、段和区域)；80130在CPU原来的指令系统的基础上，还提供了三十五条操作系统原语指令；此外，80130还提供了八个中断的硬件支持、一个系统时钟、一个延迟时钟以及一个波特率发生器。操作系统处理机的一般结构如下图所示。

图11.1中的iOSP 86软件包为应用程序与操作系统处理机之间提供了一个接口，它也可以用作简化实时多任务系统的一个开发工具。右半图与左半图的区别就在于，它在操作系统处理机的基础上增加了I/O管理、人机接口这样一些较高级的操作系统功能。实际上我们可以把80130看成是用硬件实现了iRMX 86操作系统核心部分的功能。

归纳一下，可以说操作系统处理机iAPX 86/30、iAPX 88/30具有以下特征：

(1) 包含操作系统原语的高性能的双片数据处理机；

(2) 包括iAPX 86/10、88/10的标准指令集，还具有一些任务管理、中断管理、信息传送管理，同步控制以及存贮器分配原语指令；

(3) 可扩充到iRMX 86，且与iRMX 86操作系统完全兼容；

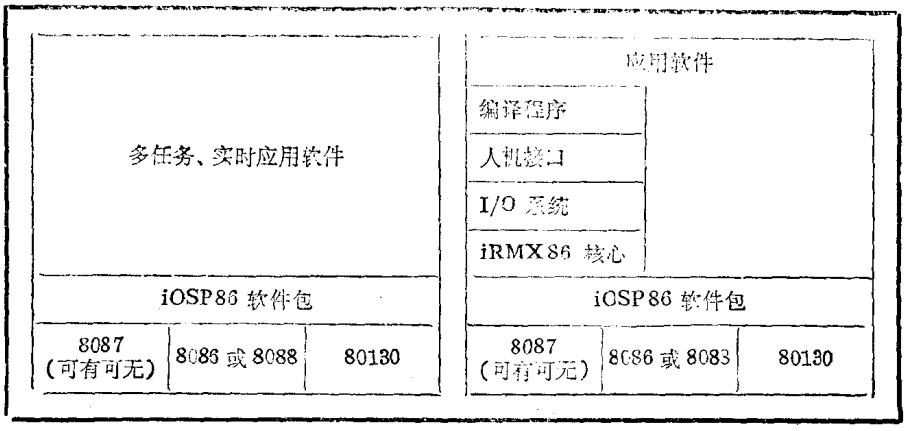


图 11.1 典型系统的结构

(4) 支持五种操作系统数据类型，它们是：作业(JOB)、任务(TASK)、段(SEGMENT)、信箱区(MAILBOX)和区域(REGION)；

(5) 提供了三十五条操作系统原语指令；

(6) 具有一些内部操作系统计时器以及可扩充到 8 到 57 种中断的中断控制逻辑；

(7) 与 8086/8087/8088 兼容；

(8) 与 MULTIBUS 兼容的接口。

§11.2 操作系统固件 80130

通过前面几章的介绍，我们已经知道 8089 输入/输出处理器是专为承担 8086 系列系统中的输入/输出工作而设计的；数值处理器 8087 是一个专用于进行数值运算的 8086 系列的协处

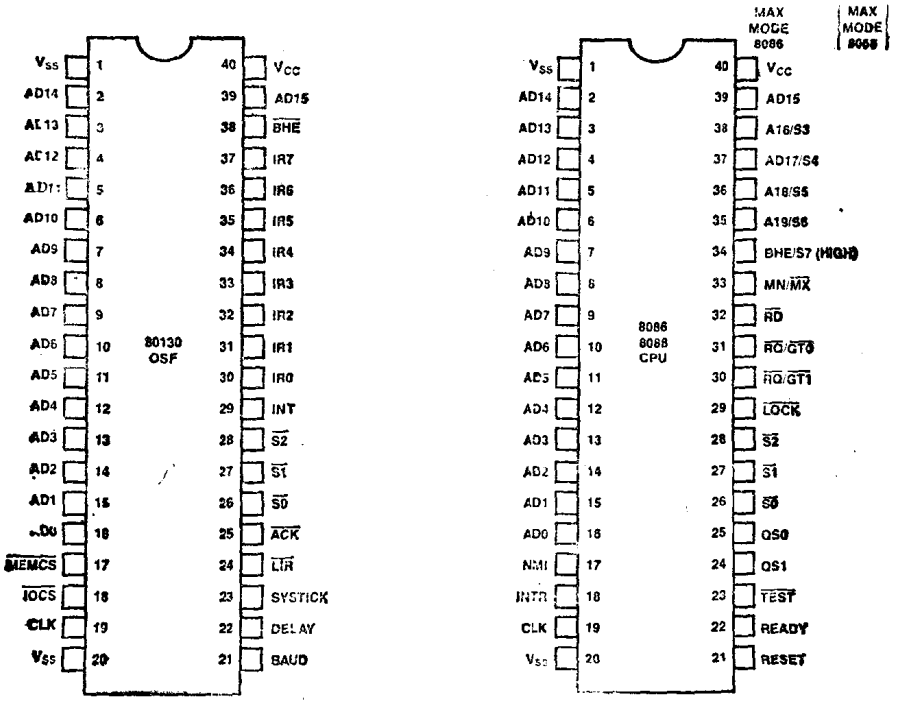


图 11.2 操作系统处理机的引脚结构

理器；而操作系统固件 80130 则从硬件上实现了操作系统的核心功能，为在 8086 系列机上实时多任务系统的实现提供了一个极为方便的工具。下面，我们准备先简单地介绍一下 80130 的诸引脚及其功能。

AD15—AD0 输入/输出信号

地址/数据线，它们的定义类似于 8086 中相应信号的定义，即为分时多路公用的存贮器地址（T_1）和数据（T_2, T_3, T_w, T_4）总线。总线周期的 T_1 间所给出的地址将被内部锁存，如果 MEMCS 信号或 IOCS 信号对所引用的原语是有效的，那么就把地址线上的内容解释为 80130 的内部地址。

$\overline{BHE}$/S7 输入信号

总线高字节数据可用信号，80130 通过利用来自 CPU 的 $\overline{BHE}$ 信号，确定是响应数据总线上的高字节部分，还是响应数据总线上的低字节部分，或是响应整个字（高字节和低字节）。该信号是这样控制 80130 的输出数据的：

$\overline{BHE}$	A0	
0	0	字送到 AD15—AD0 上
0	1	高字节送到 AD15—AD8 上
1	0	低字节送到 AD7—AD0 上
1	1	高字节送到 AD7—AD0 上

$\overline{S2}$, $\overline{S1}$, $\overline{S0}$ 输入信号

这三条引脚代表状态信息，对 80130 来说，这些状态信号仅可以用作输入，其编码如下：

$\overline{S2}$	$\overline{S1}$	$\overline{S0}$	
0	0	0	INTA（中断响应）
0	0	1	IORD（I/O 读）
0	1	0	IOWR（I/O 写）
0	1	1	无源
1	0	0	取指
1	0	1	MEMRD（存贮器读）
1	1	×	无源

CLK 输入信号

时钟输入信号，系统时钟为处理机和总线控制器提供了基本的定时信号，80130 将系统时钟用作为 SYSTICK 和 BAUD 计时器的输入，并使它们的操作能与主机 CPU 同步进行。

INT 输出信号

这是一个中断信号，只要有一个有效的中断请求发生时，INT 就将为高电平。这个信号通常总是接到 CPU 的 INTR 引脚上，起中断 CPU 的作用。

IR7～IR0 输入信号

这是八个中断请求信号，在 80130 中，中断请求信号的产生有这样两种方式，一种是通过给某个 IR 输入端一个信号（低电平跳到高电平），并将其保持为高电平直到它得到响应；另一种方式就是直接给某个 IR 输入端一个高电平。

$\overline{ACK}$ 输出信号

$\overline{ACK}$ 代表响应，该信号能够用来表示总线准备好响应，也可以用作总线接收/发送器的控

制信号。在访问 80130 的资源时，该信号为低电平；当 80130 正在提供中断向量信息时，那么在第一个 INTA 周期和第二个 INTA 周期里，它也将取低电平。

MEMCS 输入信号

存贮器片选信号。当 CPU 正在取某个核态原语时，它就将此输入端置为低电位，此时，把 AD13—AD0 用来选择指令。

IOCS 输入信号

输入/输出片选信号。在 IORD (I/O 读)或 IOWR (I/O 写)周期当中，当 $\overline{\text{IOCS}}$ 为低电平时，则表示 80130 的核态原语正在访问某个外部功能部件，定义如下：

$\overline{\text{BHE}}$	A3	A2	A1	A0	
0	×	×	×	×	无源
×	×	×	×	1	无源
×	0	1	×	×	无源
1	0	0	×	0	中断控制器
1	1	0	0	0	系统时钟
1	1	0	1	0	延时计数器
1	1	1	0	0	波特率时钟
1	1	1	1	0	时钟控制

LIR 输出信号

LIR 是一个局部总线中断请求信号。

SYSTICK 输出信号

系统时钟，通常把它接到 IR2 上以代表操作系统的定时中断。

DELAY 输出信号

延迟时钟，目前它还未被用于任何目的，即 DELAY 信号被 Intel 公司保留，将来可能要派上用场。

BAUD 输出信号

波特率发生器的输出。

V_{CC} 电源 +5V

V_{SS} 接地

在上一节中，我们曾经提到操作系统处理机除了具有基本的数据类型之外，还支持几种系统数据类型，即作业、任务、段、信箱区及区域，显然，这是在系统中加入 80130 之后的结果。下面，简单地介绍一下这几种系统数据类型。

1．作业(JOB)

我们可以把作业看作是程序环境与资源的一个组织，所有任务都在这类环境中运行。每个具体的应用都由一个或多个作业所组成，而操作系统处理机的各种系统数据类型都包含在某个作业之中。作业之间彼此是相互独立的，但它们可以共享某些资源。每个作业都具有一个或多个任务，其中包括一个初始任务。作业占有一定的存贮器空间，作业还可以产生一些从属的子作业，这些子作业能够从它们的父辈那里借用存贮器。

2．任务(Task)

任务就是具有自己的执行堆栈与私有数据的一个指令流，计算的完成是通过执行这样一

些任务实现的。换句话说，任务也就是由操作系统内核所初始化和调度的一些执行代码段。任意一个任务都肯定是某个作业的一个部分，它们所能够使用的资源也被限制在其作业所提供的资源的范围之内。对一个任务来讲，它可以是完成某个中断处理工作，也可以是完成一些别的计算功能。每个任务都具有一组属性，这些属性由操作系统处理机所保持，它们确定了任务所处的状态，这些属性包括：

该任务所属的作业

任务的寄存器状态

任务的优先级别(0～255)

表 11.1 OSP 的原语指令表

分　类	OSP　原　语	功　能
作业管理	CREATE JOB	划分系统资源，创建一个作业(任务运行的环境)
任务管理	CREATE TASK	创建一个任务(任务状态环境)
	DELETE TASK	删去任务(即删去任务状态图)
	SUSPEND TASK	挂起指定的任务
	RESUME TASK	任务挂起深度减 1
	SET PRIORITY	改变任务的优先级别
	SLEEP	将任务置为睡眠状态
中断管理	DISABLE	禁止某个外部中断级
	ENABLE	允许某一级别的外部中断
	ENTER INTERRUPT	为某个中断处理程序设置数据段基址
	EXIT INTERRUPT	完成中断处理过程
	GET LEVEL	返回正在运行的中断处理程序的中断级别号
	RESET INTERRUPT	禁止该中断，删去相应的中断任务
	SET INTERRUPT	给某个中断处理程序一个中断级
	SIGNAL INTERRUPT	中断处理程序以此来调用相应级别的中断任务
	WAIT INTERRUPT	挂起中断任务
存贮器管理(段)	CREATE SEGMENT	从作业存贮器堆中分配一个特定长度的段
	DELETE SEGMENT	回收该段的存贮器空间
信箱区管理	CREATE MAILBOX	创建一个信箱区
	DELETE MAILBOX	删除一个信箱区
	RECEIVE MESSAGE	发出调用的任务从信箱区中接收信息
	SEND MESSAGE	发出调用的任务把信息送到信箱区中
区域	ACCEPT CONTROL	请求使用区域，但即使区域不可用，也不等待
	CREATE REGION	创建一个区域
	DELETE REGION	删去一个区域
	RECEIVE CONTROL	请求使用区域，当区域不可用时，开始等待
	SEND CONTROL	放弃一个区域
环境管理	DISABLE DELETION	保护一特定的段、任务、信箱区或区域免遭删除
	ENABLE DELETION	允许一特定的段、任务、信箱区或区域被删除
	GET EXCEPTION HANDLER	读取事故处理程序的状态信息
	GET TYPE	给出某个系统数据类型一个系统类型码
	GET TASK TOKENS	给出某个任务的标志
	SET EXCEPTION HANDLER	建立当前事故处理程序的位置及处理方式
	SET CS EXTENSION	把某个新的原语与核联结起来
	SIGNAL EXCEPTION	激活相应的事故处理程序

任务的运行状态(包括睡眠、挂起、就绪、运行、睡眠/挂起这五种状态)

挂起深度

用户所选择的异常事件处理程序

若系统包含 8087，则任务的状态属性还应包括与 8087 有关的一些状态信息。

3．段(Segment)

段是存贮器的分配单位，它是物理上连续的、以十六字节为基本单位的一个存贮器区域。当作业的某个任务请求使用存贮器时，系统就将在该作业的自由存贮器空间中创建一些段，分配给发出本次请求的任务(动态分配)，而当任务不再需要某个存贮器段时，它就释放该存贮器段空间，这个段也随之消失。操作系统处理机对自由存贮器空间的保护与管理方法是很简单、规则的，发出使用存贮器请求的任务从它所对应的作业的存贮器堆中获得需要的存贮空间，当它不再需要某个存贮器空间时，就将把该存贮器空间归还给相应的作业存贮器堆(或称为父作业堆)。倘若可用的自由存贮器空间不能满足需求，那么，就不再分配存贮器给发出请求的任务，而是返回一个指出这类错误的代码。

4．信箱区(Mailbox)

引进信箱区的目的是为了实现任务间的通信，信箱区由任务用来发送或接收信息段。操作系统处理机要为每个信箱区创建两个队列并对之进行管理，其中一个队列由已经发送给信箱区但还未被其它任何任务所接收的信息段所组成(信息段队列)，另一个信箱区队列由正在等待从某信箱区接收信息的那些任务构成(任务队列)。OSP 能够确保那些正在等待接收信息的任务尽快地收到所需的信息，即信息一旦到达，任务队列中的任务就将从信箱区中获得这些信息。由此可见，在任何时刻，信箱区队列都有为空的可能。

5．区域(Region)

区域类似于"临界区"的概念，它主要是用于实现串行化和互斥的功能。实际上，区域这种系统数据类型是由一个任务队列所组成的，任务队列中的每个任务都在等待执行某个互斥码段，或是在等待访问某个共享数据区(如修改文件记录)。

80130 提供了三十五条操作系统原语指令，表 11.1 列出了这些原语，至于各个原语指令的管理功能留待后面讨论。

§11.3 操作系统处理机 iAPX 86/30、88/30 的结构

前面曾经指出，Intel 的操作系统处理机是为了适应实时多道程序环境的需要而设计的，其设计思想就是通过提供一个由硬件直接支持的、经过严格定义的、正确的操作系统原语集合，来简化多任务应用系统的设计，从而减轻应用程序员设计多任务操作系统的负担。操作系统处理机 86/30 和 88/30 分别是由一个主处理器(8086 或 8088 CPU)与操作系统固件 80130 组成的。80130 提供了一组用于多任务环境的核态原语、核控制存贮器以及其它一些支持硬件，包括这些原语所需要的系统时钟和中断控制部件。图 11.3 画出了操作系统固件 80130 的内部结构框图。实际上，从应用程序员的角度来讲，操作系统固件 80130 是基本的 iAPX86、88 结构的扩充，它提供了三十五条操作系统原语指令，增加了五种新的系统数据类型，即在 iAPX 86/10、88/10 的基础上，增加了一个操作系统内核。

80130 与 iAPX86/10 或 88/10 是以紧密耦合的方式运行的，它位于 CPU 的局部多路总

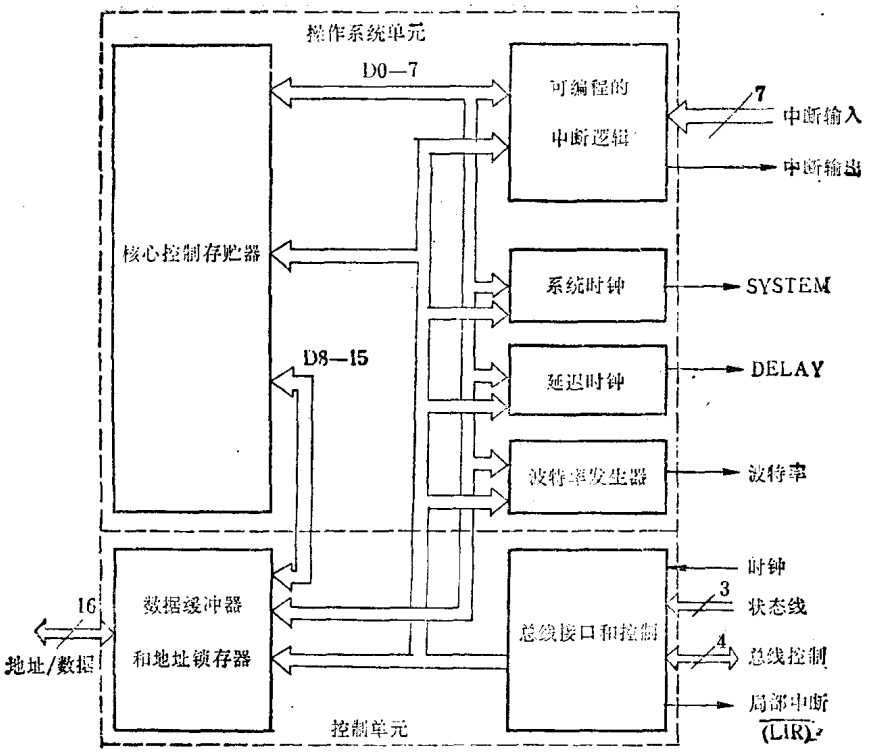


图 11.3　操作系统固件 80130 的内部框图

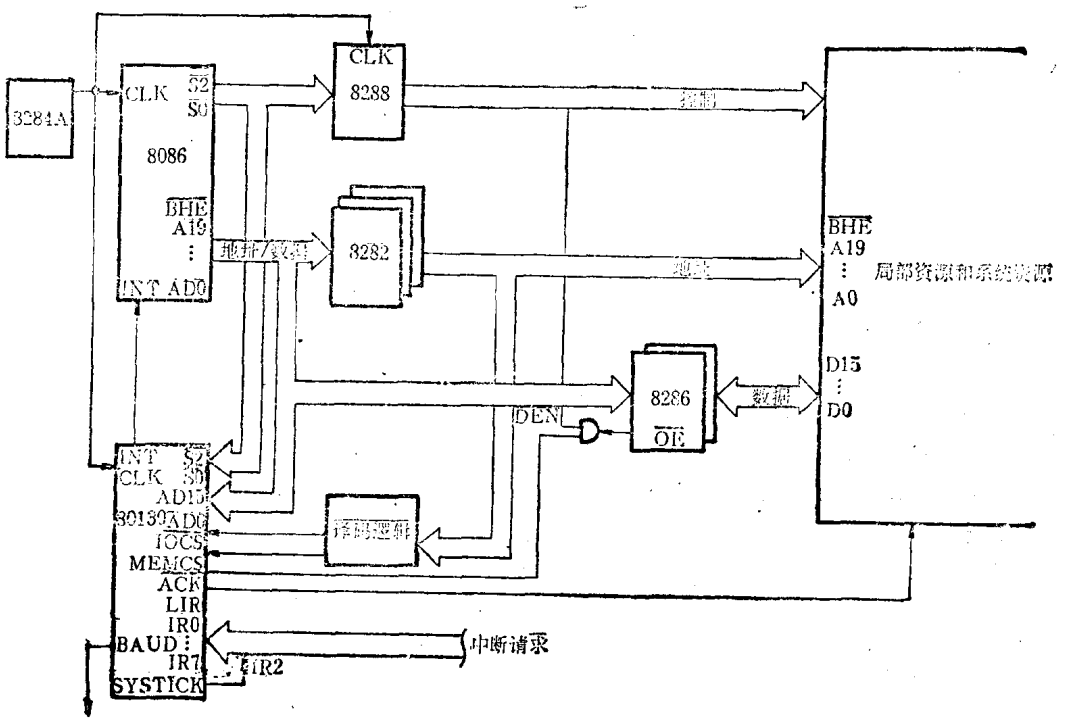


图 11.4　操作系统处理机 86/30 的典型结构

线上，其中主机总是工作在最大模式，80130 能够自动地选择是采用 86/30 操作方式，还是采用 88/30 操作方式。80130 可以用 5 MHz 和 8 MHz 两种频率进行工作，且不需要处理机的等待状态。80130 也可以与数值处理器 8087 和 I/O 处理器 8089 一起组成系统，它提供了完整的 8087 控制手段。80130 与 8086 连接起来所形成的 iAPX 86/30 的结构如图 11.4 所示，iAPX 88/30 的结构与此完全类似。

80130 既可以作为 I/O，也可以作为存贮器映象到 CPU 的局部总线上，CPU 的状态信号 (S_0～S_2) 与 $\overline{IOCS}$ 或 $\overline{MEMCS}$ 一道进行译码，从而产生相应的总线控制信号。由于 80130 的存贮器映象部分被限制在 16 K 字节的范围之内，因而，可以将地址线 A 19—A 14 用来形成 MEMCS (存贮器片选)信号，80130 能够驻留在存贮器的最高部分 (FC000H—FFFFFH) 和最低部分(00000 H—003 FFH)之外的任意一个 16 K 字节的空间之中。为了使得 80130 与多数译码逻辑设计兼容，除了上面仅有的这一点限制之外，80130 的控制存贮器代码是独立于在存贮器中的位置的。80130 的核态控制存贮器对 A 13—A 0 译码，从而形成可访问 16 K 字节的存贮器地址。至于 80130 的 I/O 映象部分则被限制在 16 个字节的范围内，故应该把地址线 A 15—A 4 用来形成 IOCS 信号，而留下 A 3—A 0 这四根地址线对 16 个字节的 I/O 空间进行寻址。

操作系统处理机有效地提高了管理系统资源的速度，表 11.2 给出了几种具有代表性的 OSP 原语的近似执行时间，被测对象是一个时钟频率为 8MHz 的 iAPX 86/30 系统。从表 11.2 中可以看出，iAPX 86/30 处理这些工作所花的时间与小型计算机很相近，而比一般的微处理机快。

表 11.2 OSP 原语的执行性能举例

数据类型	原语的执行时间(单位：微秒)	
JOB	CREATE JOB	2950
TASK	CREATE TASK	1360
SEGMENT	CREATE SEGMENT	700
MAILBOX	SEND MESSAGE (有任务切换)	475
	SEND MESSAGE (无任务切换)	265
	RECEIVE MESSAGE (任务等待)	540
	RECEIVE MESSAGE (信息等待)	260
REGION	SEND CONTROL	170
	RECEIVE CONTROL	205

由于 80130 可以用于多种不同的系统当中，因此为了使之适应于实际系统的需要，必须在初始化时规定系统的格局，指明系统中是否包含 8087，是否使用了 8259 A 中断控制器，还要设置操作系统的时钟基准，以及指出事故处理程序的控制参数等等，这样一些初始化工作都是由 iAPX 86/30 及 iAPX 88/30 的操作系统处理机支持软件包 iOSP 86 所完成的。

图 11.4 给出了一个典型的 iAPX 86/30 (或 iAPX 88/30) 系统结构，图中没有画出的只是那些很可能随应用而变化的子系统，该模型包括一个工作在最大模式的 8086 (或 8088)，一个 8284 A 时钟发生器，以及一个 8288 系统总线控制器，当然还包括一个操作系统固件 80130。

OSP 时钟总是接到数据总线的低字节上，且以偶地址寻址，这些时钟是作为两个连续的字节读出的，总是先读低字节(LSB)，后读高字节(MSB)。80130 的波特率发生器是与 8254 兼容的 (方波模式 3)，其输出端 BAUD 开始将保持为高电平状态，直到计数寄存器中装入内容时

为止，在装入计数寄存器之后，在时钟的第一个下降沿就将把内部计数器的内容送到计数寄存器，然后产生波特率脉冲．相对于 80130 的 16 字节的 I/O 空间的始址，波特率发生器的地址为 0CH(12)，时钟控制字的相对地址 0EH(14)。对大多数波特率发生器的应用而言，都要用到命令字节

0B6H （读/写波特率延时值）

一个使用 833 作为计数值，即把波特率设置为 9600 的指令序列如下所示：

```
MOV    AX, 86H        (准备把延时值写到时钟 3)
OUT    OSF+14, AX     (设置时钟控制字)
MOV    AX, 833
OUT    OSF+12, AL     (先写低字节)
SCHG   AL, AH
OUT    OSF+12, AL     (再写高字节)
```

可编程的中断控制器 (PIC) 也是 80130 的一个组合部件，它的八个输入引脚分别处理八个优先级的向量中断，其中的一个中断请求引脚必须用于定时等待功能．在初始化 80130 的过程中，将规定各个中断请求引脚的中断感应方式(水平感应或边缘感应)。如果使用了 8259A 中断控制器，那么就能够把 OSP 的外部中断数目扩充到 57 个。OSP 对中断的处理顺序是这样的：

1) 80130 的中断请求输入端上出现一个或多个中断请求信号；

2) 80130 分析这些中断请求，若适当的话，它就给 CPU 发出一个 INT 信号，请求 CPU 的干预；

3) CPU 响应该中断请求，即以一个中断响应周期作为回答，由 $\overline{S0}$、$\overline{S1}$、$\overline{S2}$ 的编码状态就可以确定它是否处于中断响应周期；

4) 一旦 80130 从 CPU 收到第一个中断响应信号，它就将设置优先级别最高的中断，并且清除相应的边缘检测锁．在这个总线周期内，80130 不驱动地址/数据总线，但它把 $\overline{ACK}$ 置为 0，并且为正得到响应的 IR 输入信号发出 $\overline{LIR}$，以证实此时处于对该中断请求的响应周期；

5) 接着，CPU 就开始第二个中断响应周期，在此周期，80130 将把该中断输入的级联地址放到总线上，根据需要还可能要把一个八位指针也放到总线上，CPU 将由此读到该指针值；

6) 这就完成了本中断周期．ISR 位在中断处理程序结束点调用一个适当的 EXIT INTERRUPT 原语 (EOI 命令)之前，将一直保持有效。

§11.4 操作系统处理机的管理功能

作为操作系统处理机，iAPX 86/30 或 iAPX 88/30 提供了许多的基本管理功能，利用它所具有的系统数据类型 (SDT) 和原语指令，就可以比较有效地分配、管理和共享处理机的资源．例如，OSP 为基本 86/10 或经过扩充的 86/20 系统 (8086 + 8087) 提供了任务管理机构，即管理由硬件寄存器组和一些软件控制信息所组成的一个任务状态图，无论任务正在运行还是不在运行，操作系统处理机都将对该任务的整个任务状态图进行管理，任务能被产生、执行，也能被置为睡眠、挂起等状态，在某个任务完成了它的使命之后，还可以动态地把它删去。

我们知道，由 OSP 所提供的实际上是 iRMX 86 操作系统的核心部心，iRMX 86 是 8086

系列处理机系统中的一个功能完全的、实时多任务的操作系统，80130 扩充了 iAPX 86（及 iAPX 88）的结构，因而，我们可以把系统建立在 OSP（iAPX 86/30 或 iAPX 88/30）的基础之上。作为操作系统的核心，OSP 可以被看成是操作系统的最低层（基础），或看成是整个操作系统的最内层，由它来完成一些基本的系统功能。这样的一个核心负责对存贮器的分配、处理机资源的分配、进程之间的通讯、中断的管理等工作进行控制。在 OSP 当中，这些功能都用硬件实现了，至于其它的一些操作系统功能则可以直接加到 OSP 之上。当然，软件（包括应用软件及系统软件）的开发工作也可以在以 OSP 为基础的系统上完成。

操作系统处理机(OSP)是一个面向事件的系统，每个事件都可以由某个响应任务，或者由一个响应任务以及与该任务紧密相关的其它一些事件来进行处理。对外部事件和中断的处理是由 OSP 中断处理程序原语完成的，即当发生中断时，就调用内部中断处理子系统，从而对中断进行处理。OSP 调度程序负责协调对多任务、多事件的管理，其调度程序采用的调度算法是抢先的优先权调度方法，它与系统时钟一道来组织、管理对各个任务的处理，从而保证先运行比较“重要”的那些任务，即以事件的重要程度为次序处理所有的事件。此外，OSP 还提供了任务间的通信原语（借助于信箱区）、互斥原语（利用区域）以及多道任务应用所需要的其它一些基本功能。

80130 一共给出了三十五条操作系统核态原语，这些原语所实现的功能是：多道任务的管理、中断处理、自由存贮器空间的管理、任务间的通讯与同步。所有这些功能都是 CPU 与 80130 一起共同行动的结果，当某个应用程序调用一条 OSP 原语时，系统就将把 CPU 的某些寄存器和栈单元用来完成某些规定动作，并将结果送给应用程序。OSP 上运行的程序可以用 ASM 86 或 PL/M-86 编制。操作系统处理机的支援软件包 iOSP 86 还为 PL/M-86 应用程序提供了一个接口库子程序，同时，该接口库子程序也提供了 80130 的结构与初始化后援，提供了完整的用户文本处理功能。为了调用 80130 的核态原语，系统使用了一个标准的、以栈为基础的调用序列。即在调用某个原语之前，必须先把其参数压到任务堆栈当中，最后一个参数在栈中的偏移量装入寄存器 SI，然后，将对应于该原语的入口码送到寄存器 AX 中，最后通过一个 CPU 软件中断引用该原语。表 11.3 列出了所有 OSP 原语及其相应的中断号、入口码、参数。

表中所使用的符号解释

JOB	系统数据类型 JOB 的标记
TASK	系统数据类型 TASK 的标记
REGION	系统数据类型 REGION 的标记
MAILBOX	系统数据类型 MAILBOX 的标记
SEGMENT	系统数据类型 SEGMENT 的标记
TOKEN	任何系统数据类型的标记
Level	中断级别号
ExcptPtr	指向异常处理代码的指针
Message	信息的标记
Ptr	对代码、堆栈等地址的指针
Seg	待装入某个段寄存器中的值
…	值参数

下面这个例子是一个具有代表性的调用某个原语的 ASM 86 序列：

表 11.3 OSP 原语

类 型	OSP 原 语	中断号	入口码(AX中)	参数(调用程序的堆栈中)
JOB	CREATE JOB	184	0100H	见80130用户手册
TASK	CREATE TASK	184	0200H	Priority, IP Ptr, Data Segment, Stack Seg, Stack Size Task Information, ExceptPtr
	DELETE TASK	184	0201H	TASK, ExceptPtr
	SUSPEND TASK	184	0202H	TASK, ExceptPtr
	RESUME TASK	184	0203H	TASK, ExceptPtr
	SET PRIORITY	184	0209H	TASK, Priority, ExceptPtr
	SLEEP	184	0204H	Time Limit, ExceptPtr
INTERRUPT	DISABLE	190	0705H	Level, ExceptPtr
	ENABLE	184	0704H	Level≠, ExceptPtr
	ENTER INTERRUPT	184	0703H	Level≠, ExceptPtr
	EXIT INTERRUPT	186	NONE	Level≠, ExceptPtr
	GET LEVEL	188	0702H	Level≠, ExceptPtr
	RESET INTERRUPT	184	0706H	Level≠, ExceptPtr
	SET INTERRUPT	184	0701H	Level, Interrupt Task Flag Interrupt Handler Ptr, Interrupt Handler DataSeg ExceptPtr
	SIGNAL INTERRUPT	185	NONE	Level, ExceptPtr
	WAIT INTERRUPT	187	NONE	Level, ExceptPtr
SEGMENT	CREATE SEGMENT	184	0600H	Size, ExceptPtr
	DELETE SEGMENT	184	0603H	SEGMENT, ExceptPtr
MAILBOX	CREATE MAILBOX	184	0300H	Mailbox flags, ExceptPtr
	DELETE MAILBOX	184	0301H	MAILBOX, ExceptPtr
	RECEIVE MESSAGE	184	0303H	MAILBOX, Time Limit ResponsePtr, ExceptPtr
	SEND MESSAGE	184	0302H	MAILBOX, Message Response, ExceptPtr
REGION	ACCEPT CONTROL	184	0504H	REGION, ExceptPtr
	CREATE REGION	184	0500H	Region Flags, ExceptPtr
	DELETE REGION	184	0501H	REGION, ExceptPtr
	RECEIVE CONTROL	184	0503H	REGION, ExceptPtr
	SEND CONTROL	184	0502H	ExceptPtr
ENVIRONMENTAL	DISABLE DELETION	184	0001H	TOKEN, ExceptPtr
	ENABLE DELETION	184	0002H	TOKEN, ExceptPtr
	GET EXCEPTION HANDLER	184	0800H	Ptr, ExceptPtr
	GET TYPE	184	0000H	TOKEN, ExptPtr
	GET TASK TOKENS	184	0206H	Request, ExceptPtr
	SET EXBEPION HANDLER	184	0801H	Ptr, ExceptPtr
	SET OS EXTENSION	184	0700H	Code, InstPtr, ExceptPtr
	SIGNAL EXCEPTION	184	0802H	Exception Code, Parameter Number, StackPtr, 0, 0, ExceptPtr

注：所有参数都将压入OSP堆栈，每个参数占一个字的空间。

```
PUSH   P1                                  (第一个参数压入堆栈)
PUSH   P2                                  (压入参数 2)
 ⋮
PUSH   PN                                  (压入第N个参数)
PUSH   BP
MOV    BP, SP
LEA    SI, SS:NUM BYTES PARAM + 2[BP]      (使 SI 指向堆栈中的第一个参数)
MOV    AX, ENTRY CODE                      (该原语的入口码装入 AX)
INT    184                                 (80130 中断)
       调用 OSP 原语
POP    BP
RET    NUM BYTES PARAMS     (弹出参数)
```

在此，CX 中含有异常事件代码；若 CX 不等于 0，表示发生过异常事件，此时，DL 中就放着引起事故的参数；AX 中放着返回值；ES:BX 中含有返回值的指针

总起来讲，OSP 有这样几个主要功能，

作业和任务管理

中断管理

存贮器管理

任务之间的通讯

任务间的同步

环境控制

§11.4.1 作业与任务管理

OSP 的一个作业实际上是一个受到控制的环境，在这种环境中可以执行一组任务。每个应用程序都在其独立的环境中运行，80130 的系统数据类型也位于该环境当中，环境要受到分配给某应用程序的资源的限制。一个应用程序通常就对应于一个单独的 OSP 作业，它可以由多个任务组成，也可以只包含一个初始任务。系统中所有的作业将系统存贮器分成一些存贮器堆。在某个作业所具有的存贮器堆中，其存贮空间将由该作业的任务状态图、其它系统数据类型所占用，从属于该作业的所有任务的运行空间也将从此存贮器堆中获得。通过管理作业内各个任务使用的资源，包括 CPU 寄存器、8087 寄存器、堆栈、系统数据类型和自由存贮器空间堆，OSP 为系统中多个任务的运行提供了基础。

系统中的所有操作都是由一些任务完成的，我们可以把任务看成是程序的执行单位，它类似于进程的概念。在创建一个任务时，OSP 首先将从其作业的自由存贮器堆中为该任务分配栈空间及数据区，并对其它的一些任务属性设置初值，即设置此任务的优先级别，规定出错处理程序的位置等等。任务优先数在整数 0 到 255 之间取值，在本系统中，优先数越低的任务其调度优先级越高，即优先数小者优先级高（这一规定与 UNIX 相同）。就一般情况来讲，0 到 128 这些优先数将给处理中断的那些任务，超过 128(129～255)的优先数适宜于那些非中断任务。倘若系统中用了数值处理器 8087，那么，给运行中的任务的出错恢复中断级别应该比任

务的优先级高，以便能够正确地对错误进行处理。

每个任务都有一个运行状态，OSP 中的任务共有五种可能的运行状态，它们是：运行(RUNNING)、就绪 (READY)、睡眠 (ASLEEP)、挂起 (SUSPENDED) 和睡眠——挂起 (ASLEEP-SUSPENDED)。

当某个任务占有处理机控制权时，我们就说它处于运行状态。

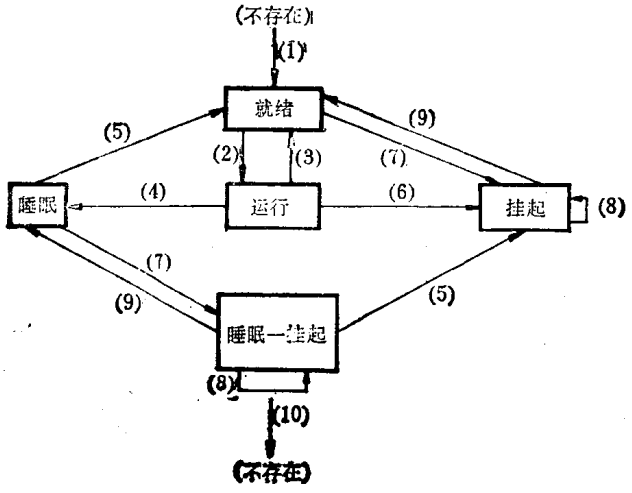
若一任务既非睡眠，也未被挂起，又不处于睡眠——挂起或运行状态，则它肯定处于就绪状态。一个任务若要变成运行状态，它就必须为就绪状态且具有最高的优先权。

若一任务正在等待对某个请求的响应，或在等待时钟事件的发生，则称该任务处于睡眠状态。任务可以把自己置为睡眠状态。

当一个任务挂起自己或被另一个任务挂起时，该任务就将进入挂起状态。对一个任务来讲，它可能被多次挂起，挂起的次数也称为任务的挂起深度，OSP 在对任务进行管理时，也要用到挂起深度这个参数。

若某个任务既在等待，又被挂起了，则说它处于睡眠——挂起状态。

图 11.5 画出了 OSP 任务的五种可能的运行状态，并且给出了任务从一种运行状态转为另一种运行状态的一些例子。系统能够保证各个任务处于正确的状态。



注：(1) 创建该任务
(2) 该任务变为优先级最高的就绪任务
(3) 该任务被另一个优先级别更高的任务挤出执行状态
(4) 该任务调用原语 ASLEEP，或等待某个事件的发生
(5) 任务睡眠时间已到，或等待过程结束
(6) 任务调用原语 SUSPEND，把自己挂起
(7) 任务被其它任务挂起
(8) 该任务被其它任务挂起，或对该任务执行一次 RESUEM，但其挂起深度仍然不等于 0
(9) 该任务被某个别的任务解挂
(10) 该任务被删去

图 11.5 任务状态图

任务的属性，包括 CPU 寄存器的值和 8087 寄存器的值(如果使用了 8087 的话)、优先数以及与之相关的一些资源，都由 OSP 保存在任务状态图当中，每个任务自身都具有一个任务状态图。

尽管系统中的多个独立的任务似乎同时在运行，然而，实际上在某一时刻只可能有一个任务真正在运行，因此，需要一些调度方法以确定处理机为哪个任务服务。OSP 中的处理机调

度是以各个任务的优先级别为基础的，每个任务相对于系统中的其它任务来讲，都有一个给定的优先级别，OSP 将各个任务的优先级分别保存在各自的任务状态图中。当某个具有较正在运行的任务的优先级为高的任务变成就绪状态时，OSP 就会将处理机的控制权让给系统中优先级最高的任务，当需要进行任务切换时，OSP 首先要把包括 CPU 寄存器在内的有关被调出任务（原在运行的任务）的状态保存在该任务的状态图中，然后，OSP 从调入任务（当时优先级别最高的任务）的任务状态图中恢复 CPU 寄存器的内容，最后，让 CPU 开始执行该优先级较高的任务。由此可见，任务调度工作是由 OSP 完成的，这种面向优先级的抢先调度策略非常简单，OSP 只要对各任务（就绪状态的任务）的优先级进行比较，就能够确定让哪个任务运行，从而可以保证系统中处于就绪状态的优先级别最高的任务投入运行。一般来说，一旦某个任务开始运行，它将继续执行下去，除非在其运行期间，发生了一个优先级更高的中断，或该任务请求使用一个暂时不能为之服务的资源（此时它将准备等待），或该任务使得某个特殊的资源能够用于某个正在等待此资源的优先级更高的任务（进行任务切换操作），在以上这三种情况下，当时处于运行状态的任务将改变运行状态，让出处理机。

OSP 时钟硬件提供了定时等待和时间输出功能，正因为如此，在许多原语当中，一个任务能够确定它准备等待某件事件发生的时间长度，或为等待所需资源变为可用或是在某信箱区中接收到信息所准备等待的时间。时间间隔可以被修改，但下限为 1 毫秒。

由于有操作系统（或 OSP）负责任务调度工作，因而，每个任务都可以认为专门有一个处理机为它服务。任务可能发出象接收信息之类的系统调用，这就有可能导致另一个任务投入运行，因为当时也许没有可以接收的信息。此外，中断也很有可能使一个不同的任务投入运行。实际上，系统中也只有中断或系统调用会导致优先级更高的任务变为就绪状态，故当发生中断或系统调用时，就将执行一次任务调度工作，让优先级最高的任务开始运行。这种调度过程是通用的，所以在把新的任务加到系统中去时，无需修改调度过程。对于一个“好”的调度系统来说，系统中的所有任务都可以比较均衡地占用处理机，可以给用户一个这些任务在同时运行的感觉。

程序还可以动态地改变任务的运行状态及优先级别，一个任务可以将本身或另一个任务挂起一段时间，以后再继续执行，已经被挂起的任务可以被再次挂起。OSP 中的八个作业和任务管理原语是：

CREATE JOB：划分系统资源，为一组任务创建一个运行环境，同时，生成一个初始任务及其堆栈。

CREATE TASK：创建一个任务状态图，规定该任务指令码的位置、运行优先级、堆栈以及其它一些任务属性。

DELETE TASK：去除指定任务的状态图，从系统中以及有些等待队列中删去该任务，停止执行对应的指令流，收回分配给它的资源。但需注意，不能删除中断任务。

SUSPEND TASK：挂起指定的任务，倘若该任务已经处于挂起状态，则将其挂起深度加 1。执行此原语后，任务的运行状态将成为挂起状态。

RESUME TASK：任务挂起深度减 1。若减 1 之后，任务的挂起深度仍不等于 0，则该任务的运行状态不变；否则，本原语将把此任务的运行状态置为 READY（就绪）或 ASLEEP（睡眠）。究竟置为何种状态则是由执行本原语之前该任务所处的运行状态所决定的，若先前该任务处于挂起状态，则执行本原语之后，它将成为就绪状态；若该任务原为挂起——睡眠状态，则

执行该原语之后，它将成为睡眠状态。

SLEEP：将任务置为睡眠状态，并规定睡眠时间的长度，所规定的睡眠时间最长为10 ms。

SET PRIORITY：改变任务的优先级别。

§11.4.2 中断管理

OSP最多可以支持256个中断级，由它们构成了一个向量中断。OSP最多可以处理57个外部中断源，其中之一为不可屏蔽中断(NMI)。OSP独立地对各个中断级进行管理，它的中断子系统提供了两种中断处理机构，即中断处理程序(INTERRUPT HANDLER)和中断任务(INTERRUPT TASK)。中断处理程序在执行时屏蔽掉了所有的可屏蔽中断，中断处理程序是一个很短的过程，它仅有的功能就是尽可能快地对中断作出响应，正是为了能使中断处理程序以最快的速度运行，因而屏蔽掉了所有的中断。由此可见，中断处理程序仅仅应该用于那些只需很少处理时间的中断服务。在中断处理程序中，只能使用80130的一部分中断管理原语(如DISABLE、ENTER INTERRUPT、EXIT INTERRUPT、GET LEVEL、SIGNAL INTERRUPT)和CPU的一些基本指令，而不可以使用另外的一些OSP原语。但是，中断任务却与此不同，它可以使用所有的OSP原语，且在其执行时仅可屏蔽优先级别较低的中断。

处于同一级别上的中断处理程序与中断任务之间的工作转移是通过原语SIGNAL INTERRUPT和WAIT INTERRUPT实现的，所采用的是异步转移方式。当某个中断处理程序向对应的中断任务发出一个信号之后，该中断处理程序就可转去处理另外的一个中断。对某个特定的中断级而言，中断处理程序能为中断任务提交的中断数目可以在原语SET INTERRUPT中规定，等待中断任务处理的所有中断都排在相应的中断队列中，在当前的中断任务完成时，它就发出一个WAIT INTERRUPT原语，并准备立即转去处理中断队列，此中断队列是在该中断任务处理过程中，由对应的中断处理程序重复使用SIGNAL INTERRUPT原语而建立起来的。倘若该中断级上不存在中断，那么，其中断队列就肯定为空，这时也就应该挂起相应的中断任务。图11.6给出了中断处理的一般流程，而图11.7则给出了一个比较具体的例子。

OSP中的外部中断级别是直接与内部的任务调度优先级别相关联的，OSP中有一张包含所有的任务和中断的优先级表格。处于运行状态的任务之优先级自动地确定了应该屏蔽掉哪些中断，即仅仅允许那些优先级更高的中断出现，也只有这些中断能够中断当前任务的执行。中断的管理是通过中断级别实现的。OSP可以直接支持八级中断，假如加上8259A中断控制器，它就能够支持多达五十七级中断。

OSP中有九条用于中断管理的原语，它们是：

DISABLE：禁止某个外部中断级。

ENABLE：允许外部中断级。

ENTER INTERRUPT：给某个中断处理程序一个属于它自己的数据段，该数据段是与被中断的任务的数据段分开的。

EXIT INTERRUPT：执行一次“中断结束”动作。该原语是由未引用中断任务的那些中断处理程序使用的，利用它向硬件发出中断结束信号。当中断处理程序执行完此原语之后，它就将放弃处理机的控制权，从而也相当于允许那些可屏蔽的中断发生了。

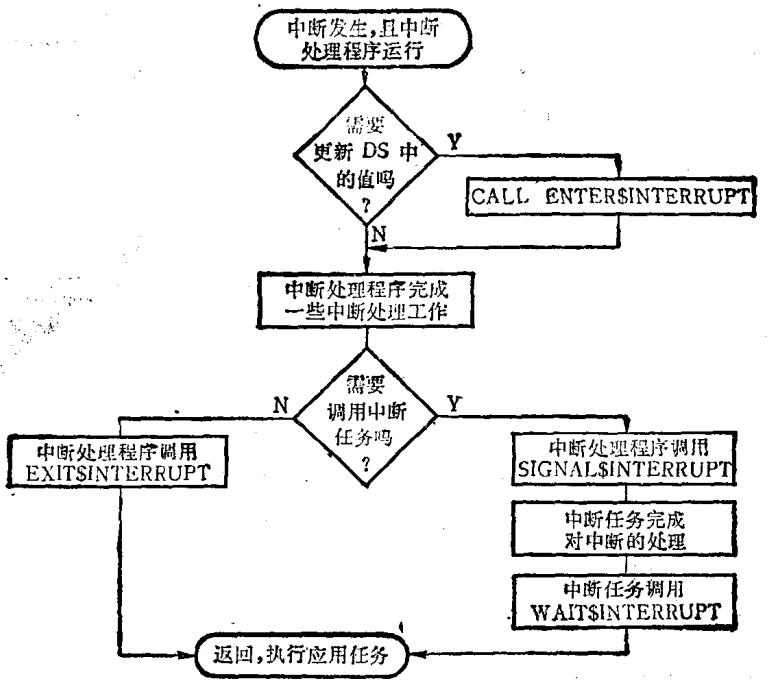
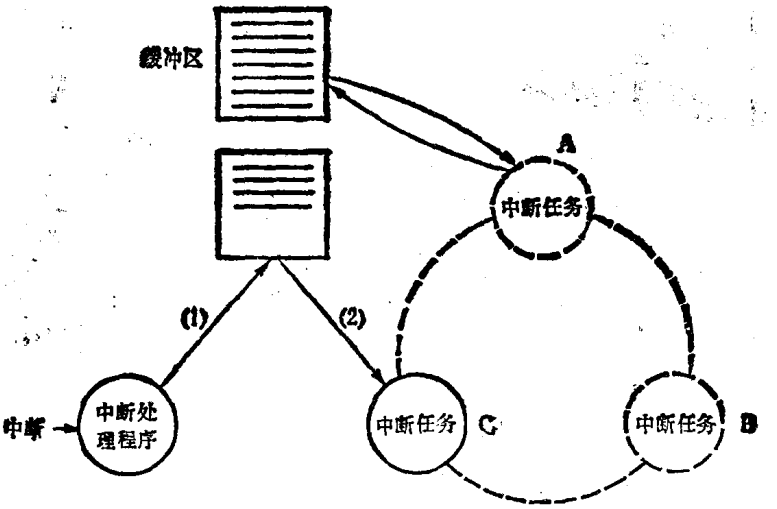


图 11.6 中断处理流程



(1) 开始向空的缓冲区装入信息　(2) 缓冲区已满,调用 SIGNAL$INTERRUPT

A 从已满的缓冲区中获得信息　B 对满的缓冲区进行处理

C 调用 WAIT$INTERRUPT等待下一个缓冲区装满

图 11.7 多个缓冲区举例

GET LEVEL：返回正在运行的中断处理程序所对应的中断级号，若有几个中断处理程序同时在运行，则返回优先级最高的中断处理程序的中断级别。

RESET INTERRUPT：禁止该中断级，删去其中断处理程序，若对应于该级别有中断任务存在，则也将删除它们。

SET INTERRUPT：给某个中断处理程序一个中断级别，若存在中断任务，则也给它们一个同样的级别。

SIGNAL INTERRUPT：中断处理程序用此原语来调用同一级别上的中断任务。

WAIT INTERRUPT：挂起调用该原语的中断任务，直到同级的中断处理程序执行 SIGNAL INTERRUPT 原语为止，SIGNAL INTERRUPT 将唤醒相应的中断任务。若在 WAIT INTERRUPT 原语执行之前，已经执行过 SIGNAL INTERRUPT，即中断队列非空，则将在原语 WAIT INTERRUPT 执行之后，立即执行中断任务，对该中断进行处理。

§11.4.3 存贮器管理

计算机系统中最为重要的资源之一就是存贮器。在有些系统当中，存贮器的需求量是在设计系统时就规定了的，但在多数应用当中，存贮器空间的需要量都是随应用的不同而变化的，因此，设计系统时就必须以最大的存贮器空间需要为准。一个比较灵活且较经济的解决办法就是根据实际的需要从一个集中的存贮器堆中分得存贮器，当不再需要时就将它归还给存贮器堆。这样做有两个优点，第一，总的存贮器空间需要量减少了；第二，这也为程序员提供了方便，因为这时程序员不再需要了解有关存贮器的许多细节。基于这样的思想，OSP 的自由存贮器空间管理主要就是管理存贮器堆，每个存贮器堆都是分配给某个作业的运行空间。执行原语 CREATE JOB 时，所创建的作业从其父作业的存贮器堆中分得一个新的作业存贮器堆，存贮器堆是作业资源中的一个部分，但它开始并没有在作业的诸任务之间进行分配。当在作业内部创建一个任务、信箱区或区域等系统数据类型的结构时，OSP 就从该作业的存贮器堆中为所需建立的结构分配存贮器，而不需要专门调用存贮器分配程序。这类隐含地用到自由存贮器管理的 OSP 原语包括 CREATE JOB、CREATE TASK、DELETE TASK、CREATE MAILBOX、DELETE MAILBOX、CREATE REGION 以及 DELETE REGION。当某个任务需要一个存贮器段时，可以通过使用 CREATE SEGMENT 原语显式地分配一块存贮空间，存贮器的段长是十六字节的整数倍，取值在十六字节到 64K 字节之间，当然，程序员可将段长规定为 1 到 64K 字节中的任何值，但 OSP 会将它们调整到适当的值（大于或等于规定长度的第一个十六的倍数）。存贮器段是用于任务堆栈、数据存贮器、系统缓冲器、程序区及任务间的信息交换区的系统存贮器的基本单位。

OSP 中有两条专门用于进行存贮器管理的原语，它们是：

CREATE SEGMENT：从作业存贮器堆中分配一个特定长度的段（以十六个字节长为单位）。

DELETE SEGMENT：释放该段的存贮空间，并将其归还到作业的存贮器堆中。

§11.4.4 任务间的通信、同步与互斥

OSP 内部具有任务间的同步与通信机构，它使得任务之间可以传输、共享信息，OSP 的信箱区具有受到控制的信息交换能力，它可以保证一个完整的信息总是由发送任务发出，且由适当的接收任务所接收。每个信箱区都由两个互锁队列组成，即一个任务队列，一个信息队列。OSP 共提供了四个用于实现任务间的同步和通信的原语指令，即：

CREATE MAILBOX：创建一个信箱区，即建立任务间的信息交换。

DELETE MAILBOX：删去一个信箱区，并将其所占的存贮空间归还给作业存贮器堆，同

时也将去除该信箱区上的信息队列。也就是说，该原语所起的作用是解除某些任务之间的信息交换能力。

RECEIVE MESSAGE：任务已准备好从信箱区中接收信息，即把调用该原语的任务放到信箱区的任务队列中。

SEND MESSAGE：发送一个信息至信箱区。

原语 CREATE MAILBOX 创建一个信箱区用于任务之间的信息交换，当发出 SEND MESSAGE 原语时，OSP 就将以 FIFO（先进先出）的方式将信息送到特定的信箱区中，完全类似地，当某任务发出 RECEIVE MESSAGE 原语指令时，OSP 就将检查是否存在该任务所要接收的信息，当然，其它的一些任务也可以使用这个信箱区，即一个信箱区可由多个任务所公用。倘若在执行 RECEIVE MESSAGE 原语时，不存在可供接收的信息，那么，企图接收信息的那个任务就开始等待。任务队列的管理方法决定了信箱区任务队列中的哪个任务将从该信箱区中取得信息，其管理方法有 FIFO 和优先级这样两种。

相对于一般的同步算法和查询任务来说，信箱区方法有它一定的优点。第一，系统中各进程之间的同步都是由某些段中的数据是否可用所确定的，而不需要其它的算法以及代码，也无需关心各任务执行速度的快慢；第二，速度较低的输入输出工作可以通过加入缓冲区来进行；第三，任务与信箱区之间的接口只是一些段，所以，任务的加入与删除都比较容易。总之，信箱区使得任务之间的通信变得很清晰、规则，为实际应用带来了方便。任务间的信息交换可用图 11.8 表示。

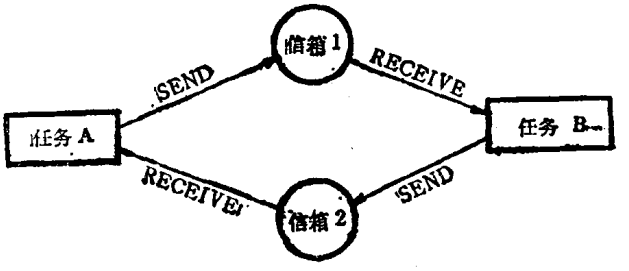


图 11.8　任务之间的通信与同步

对多道程序和多重处理系统来讲，由于系统中多个任务的存在，资源的竞争就成为不可避免的了，因此，实现任务间的互斥功能也就变得极为重要。在 OSP 中，我们用区域这种系统数据类型来实现互斥，它类似于通常所说的临界区的概念。一个区域实际上就是由正在等待使用某种资源的任务所组成的一个任务队列，其中的资源每次只能由一个任务所使用。操作系统处理机提供了一些区域控制原语，借助于这些原语就可以对互斥数据及互斥资源进行管理，利用它们，能够实现对互斥码段以及一些共享的数据机构的保护，以防止多个任务同时使用它们。区域中任务队列的排队规则有两种，即 FIFO 规则和优先级排队规则。系统中可以存在多个区域。值得注意的是，当任务位于某个区域中时，那么无论该任务自身，还是其它任务都不能挂起这个任务。因而避免了死锁发生的可能性。

用于实现互斥功能的五个 OSP 原语以下：

CREATE REGION：创建一个区域，并为之规定任务的排队规则。

SEND CONTROL：放弃一个区域。

ACCEPT CONTROL：请求获得区域的控制权，但即便是区域当时不可使用，任务也不等待。

RECEIVE CONTROL：请求获得区域的控制权，当区域当时不可使用时，任务就将开始等待。

DELETE REGION：若区域不在使用过程中，则删去它。

在遵守优先级排队规则的区域当中，OSP 还具有动态地修改任务优先级别的能力。具体一点讲就是，假如在一个任务使用某个区域时，一个具有更高优先级的任务发出 RECEIVE CONTROL 原语，请求使用同一个区域，那么，OSP 就暂时将占有区域的那个任务的优先级置为发出请求的任务的优先级，即临时提高原任务的优先级别，直到该任务通过 SEND CONTROL 放弃对区域的使用，由于这时说明它不再需要使用此互斥资源，故将该任务的优先级恢复，使它获得自己真正的优先级。也就是说，等待队列中的各个任务按照优先级的高低使用区域，而正在使用区域的任务总具有等待队列中各任务的最高优先级，从而可以保证，一旦某个任务占用了区域，那它将一直用下去，直到用完主动放弃时为止。

§11.4.5 其它控制功能

除了上面已经介绍过的 OSP 功能之外，OSP 还提供了一些对多任务系统起控制、定性作用的系统原语。借助于这样一些原语，就能够控制某些系统数据类型的去除，控制系统中自由存贮器的恢复，可以询问操作系统的状态，也为增加用户的系统数据类型提供了手段。

对每种系统数据类型来讲，它们都有一个删除属性，应用程序员通过设置某种系统数据类型的删除标志，就能够显式地控制该 OSP 系统数据类型是否可被删去。在任务、信箱区、区域和段这些系统数据类型刚产生时，其删除标置都是可以设置的，即说明它们都可以被删去。控制删除属性的两个 OSP 原语是：ENABLE DELETION 和 DISABLE DELETION。我们应该根据具体应用的需要，确定是否设置这些删除标志。

OSP 还可以进行一些询问与控制操作，这些操作手段有助于用户检查系统环境，进行灵活的事故处理。OSP 提供了五种环境控制原语。

OSP 结构的另一个特色就是：允许将新的由用户所定义的系统数据类型，以及对这些类型进行处理的原语加到原有的 OSP 功能上去。我们把为用户系统数据类型而建立的类型管理程序称为用户 OS 扩充，通过原语 SET OS EXTENSION 就可以将这些管理程序安置到系统中去。这样，类型管理程序的功能就可以用与 OSP 接口相一致的用户原语来引用了。对一个结构良好的待扩充的结构来讲，每个 OS 扩充都应该支持一种用户所定义的系统数据类型，并且要求每个 OS 扩充都给用户提供一个与原有的系统数据类型相同的调用顺序及程序接口，即要求这些新加的类型管理程序适合应用的需要。OSP 的中断向量入口 (224—255) 就是专门保留用于用户 OS 扩充的。在赋与某个用户 OS 扩充一个中断号之后，用户就能以标准的 OSP 调用次序引用该扩充的功能。

ENABLE DELETION：允许删除一特定的段、任务、信箱区或区域。

DISABLE DELETION：保护一特定的段、任务、信箱区或区域，使其免遭删除。

GET TYPE：给出某个系统数据类型的系统类型码。

GET TASK TOKENS：给出任务的标记。

GET EXCEPTION HANDLER：读出某任务当前的 OSP 事故处理程序的位置以及处理方式。

SET EXCEPTION HANDLER：为任务当前的 OSP 事故处理程序设置其位置及处理方式。

SET OS EXTENSION：修改中断向量的某个入口 (224—255)，使其指向某个 OS 扩充过

程。即把新的原语指令与操作系统的内核联系起来。

SIGNAL EXCEPTION：用于 OS 扩充的出错处理过程中。

§11.5 应 用 举 例

为了说明 OSP 原语的使用方法，我们举了下面这个例子。本例以一简化的方式跟踪一天的时间。假设系统使用 60 Hz 的 A．C 信号作为时间基准，电源所提供的是与 TTL 兼容的信号，该信号在每个 A．C 周期都要驱动一个 80130 边缘触发的中断请求引脚，中断处理程序响应这些中断，记录每秒的 A．C 周期数，中断任务对秒数计数，并在一天到后删除自身。中断处理程序与中断任务是作为两个分离的程序模块而编制的。程序如下：

```
DECLARE SECOND$COUNT BYTE,
  MINUTE$COUNT BYTE,
  HOURS$COUNT BYTE;
TIME$TASK: PROCEDURE;
  DECLARE TIME$EXCEPT$CODE WORD;
  AC$CYCLE$COUNT = 0;
  CALL RQ$SET$INTERRUPT (AC$INTERRUPT$LEVEL, 01H,
    @AC$HANDLER, 0, @TIME$EXCEPT$CODE);
  CALL RQ$RESUME$TASK(INIT$TASK$TOKEN, @TIME$EXCEPT$CODE);
  DO HOUR$COUNT = 0 TO 23;
    DO MINUTE$COUNT = 0 TO 59;
      DO SECOND$COUNT = 0 TO 59;
        CALL RQ$WAIT$INTERRUPT(AC$INTERRUPT$LEVEL,
          @TIME$EXCEPT$CODE);
        IF SECOND$COUNT MOD 5 = 0
          THEN CALL PROTECTED$CRT$OUT(BEL);
        END;（秒计数循环）
      END;（分计数循环）
    END;（小时计数循环）
  CALL RQ$RESET$INTERRUPT (AC$INTERRUPT$LEVEL,
    @TIME$EXCEPT$CODE);
  END TIME$TASK;
```

以上这个中断任务的作用就是跟踪一天的时间。下面这段代码是作为中断处理程序处理的。

```
DECLARE AC$CYCLE$COUNT BYTE;
AC$HANDLER: PROCEDURE INTERRUPT 59;
  DECLARE AC$EXCEPT$CODE WORD;
  AC$CYCLE$COUNT = AC$CYCLE$COUNT + 1;
  IFAC$CYCLE$COUNT > = 60 THEN DO;
```

```
    AC$CYCLE$COUNT = 0;
    CALL RQ$SIGNAL$INTERRUPT(AC$INTERRUPT$LEVEL,
      @AC$EXCEPT$CODE);
    END;
  END AC$HANDLER;
```

中断处理程序实际上是为中断 59 服务的，它所完成的动作仅仅是简单地对中断计数，当计数值达到 60 时，它就调用 SIGNAL INTERRUPT 原语，以通知中断任务一秒已到。利用 SET INTERRUPT 原语，把中断级号 59 赋与中断处理程序 AC$HANDLER，然后，中断任务执行原语 RESUME TASK，以便恢复应用任务的执行。

中断任务的主要部分就是一个计数循环。在中断处理程序中的 SIGNAL INTERRUPT 原语向中断任务发出信号（中断级为 AC$INTERRUPT$LEVEL），当中断任务接收到中断处理程序的信号时，它就执行一次循环，增加一下时间计数变量的值，接着，中断任务就执行原语 WAIT INTERRUPT，等待中断处理程序再次送来信号，一般来讲，该中断任务总要花一段时间等待下一个信号的到来。在一天结束时，中断任务将跳出循环，并且执行原语 RESET INTERRUPT，这就相当于禁止了相应的中断级，且删去了该中断任务，这时，OSP 就收回由该任务所使用的存贮空间，且去调度另一个任务，让其投入运行。

思　考　题

11.1　80130 的主要功能是什么？为什么设计这样一个芯片？

11.2　80130 提供了哪些系统数据类型？

11.3　何谓作业，何谓任务？80130 是如何进行作业与任务管理的？任务有哪几种运行状态？一个任务怎样才能投入运行？

11.4　80130 是如何对中断进行管理的？其中断管理机构有何特色？

11.5　简述段的概念，80130 是如何管理存贮器资源的？有何优点？

11.6　何谓信箱区，何谓区域？80130 的通信机构有何优点？80130 是如何对互斥资源进行管理的？

11.7　80130 的设计思想对微型机系统结构的发展有些什么指导意义？

第十二章　iAPX 86/88 系列

§12.1　iAPX88 微处理器

§12.1.1　概　　述

8088 是一种 8 位微处理器，它是在 8085 和 8086 的基础上发展起来的，是 8 位微处理器与 16 位微处理器的优点相结合的产物。8088 的内部采用的是 16 位结构，故也有人把它称为准 16 位的微处理器，这就使之与 8086 16 位微处理器具有相同的指令系统，大大提高了机器的性能；而外部却采用 8 位数据通路汇集数据，以便于和大多数外部设备相连接，8088 的总线功能基本上与 8085 A 相同。由此可见，8088 具有 8 位和 16 位微处理器的特征，它是与 8086 的软件和 8080/8085 的硬件及外设直接兼容的，这样，8088 就把 16 位微处理器结构与常用的 8 位存贮器和系统设计的价格优点结合起来了。此外，由于 8088 的数据通路宽度是 8 位的，故它很宜于用在诸如事务处理之类的面向字节的系统当中。这里值得提一下的是，尽管 8088 是 8 位的微处理器，似乎比不上 16 位的 8086，但由于它从系统的角度综合了 8 位及 16 位微处理器的优点，故在实际的微型机系统当中，8088 用得相当普遍，特别是 IBM 的个人计算机也采用了 8088 作为 CPU，这就更使得 8088 的身价倍增，引起了广大系统设计者和用户的极大兴趣。我们可以把 8088 的结构近似表示如下：

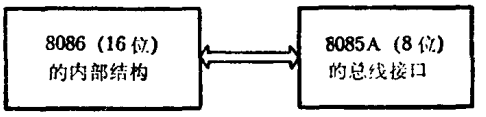


图 12.1　8088 的结构示意图

归纳起来，8088 具有如下特色：

1）8 位数据接口总线；

2）16 位的内部结构；

3）一兆字节的存贮器直接寻址能力；

4）与 8086 的软件直接兼容；

5）十四个 16 位的寄存器；

6）二十四种操作数寻址方式；

7）字节、字及字组操作；

8）以 2 进制或 10 进制表示的 8 位和 16 位数据，以及对这些数据所进行的运算，包括乘法和除法操作，其中的数据可以是带符号的，也可以是不带符号的；

9）与 8155-2、8755A-2 及 8185-2 兼容；

10）两种时钟速率：5 MHz（用于 8088）

8 MHz（用于 8088-2）。

§12.1.2　8088 的结构

图 12.2 给出了 8088 的引脚结构，从图中可以看到，8088 的引脚结构基本上是与 8086 相

同的，实际上，这些引脚的功能也与 8086 相应引脚的功能大致一致，故这里就不再重复了，至于它们之间的区别，我们将在后面介绍。

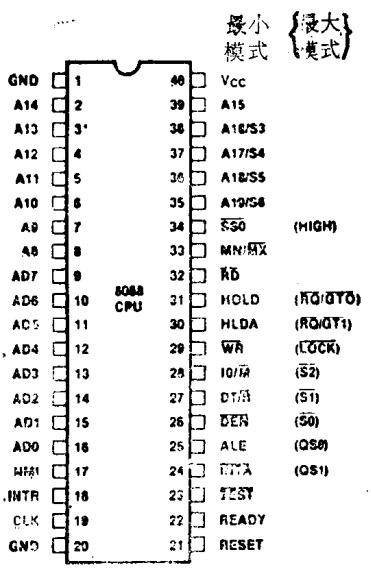


图 12.2 8088 的引脚结构

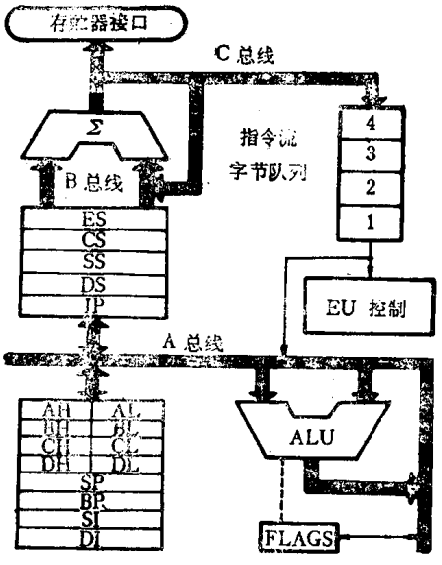


图 12.3 8088 的原理框图

根据 8088 内部各部分所起的作用的不同，我们可以画出 8088 的内部结构框图，如图 12.3 所示。

一、寄存器组

1．通用寄存器组

8088 有四个 16 位的通用数据寄存器，即 AX、BX、CX、DX，它们可按字节或字来进行寻址，若把它们当字节寄存器使用，则记为 AH、AL、BH、BL、CH、CL、DH、DL。

2．指示器和变址器组

8088 有两个指示器：堆栈指示器 SP 和基址指示器 BP；两个变址寄存器：源变址器 SI 和目标变址器 DI。

3．段寄存器组

8088 有四个 16 位的段寄存器，分别指向当前的四个可寻址存贮器段，四个段寄存器记为：CS（代码段）、DS（数据段）、SS（堆栈线）和 ES（附加段）。所有的取指操作都是在当前代码段中进行的，在执行取指操作时，指令指示器（IP）的值总是自动地与 CS 的内容相加，从而形成指令的地址；在执行堆栈操作时，栈段寄存器 SS 的内容总是自动地与逻辑地址（SP 寄存器中的内容）相加，形成堆栈单元的地址；数据操作时可以选择 DS 或 ES 作为基址，基地址加上逻辑地址就形成了实际地址，逻辑地址的确定方法（寻址方式）多种多样，寻址方式将在后面介绍，逻辑地址是 16 位的，故利用它可以访问任一给定的段空间（因为最大段长为 64 K 字节），当总的存贮器容量不超过 64 K 字节时，可以将所有的段寄存器置成同一数值，即让所有的段完全重叠。借助于这组段寄存器，就可以比较方便地实现程序的再定位和重入，这是对操作系统的一个有力支持。

4．标志寄存器

标志寄存器的主要作用就是表示 8088 所处的状态，它的格式如图 12.4 所示。

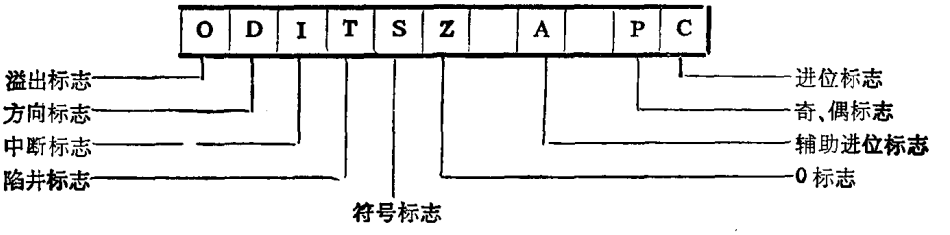


图 12.4 标志寄存器的格式

标志寄存器中 S、Z、A、P、C 的意义与 8080/8085 中的规定相同；中断标志 I 指示外部中断，当陷井标志 T 为 1 时，处理器就将采用单步执行方式，以便进行程序调试，方向标志 D 主要用于字符串操作指令。

二、8088 中的流水操作

由图 12.3 可知，8088 CPU 的内部可被分成两个部分：执行部件 (EU) 和总线接口部件 (BIU)。总线接口部件 BIU 负责取指令，以及实现执行部件 (EU) 与外部的数据传输；而执行部件 EU 则负责解释、执行指令。当 EU 还在执行前面的指令时，BIU 就将试图取出下面的指令，并把它们放入指令流队列之中。一般来讲，当 EU 执行完一条指令时，下一条待执行的指令已经被取出，且位于指令流队列的头上，因此，这时 EU 就可以直接取出并执行该指令。8088 的指令流字节队列共包含四个字节，它起着一个先进先出(FIFO)缓冲区的作用。

当跳转指令执行成功时(发生跳转)，已取入指令字节队列的那些指令就被宣告无效，EU 这时就不再是从指令流队列中取出指令，而是把转移指令的目标地址送给 BIU，让 BIU 按此地址取出应该执行的指令，EU 要等到这条指令取出之后才继续执行，然后，8088 又开始新的流水操作。

采用这种流水结构之后，BIU 一旦发现总线空闲，就将试图预取指令，显然，这使得总线得到了更为充分的利用，提高了 8088 的性能，由于取指操作和执行指令的过程几乎是并行的，从而使得 8088 具有相当于 16 位数据传输的微处理器(如 8086) 的性能。从表面上看，这种流水结构增加了指令队列及预取逻辑，似乎会提高系统的价格。但是，我们应当看到，采用流水结构之后，EU 很少需要等待 BIU 取指令，从而使系统不要求很高的数据存取速度，即系统并不要求高速的存贮组件，而可以使用速度稍慢但价格较便宜的存贮器器件，使整个系统的性能价格比得到提高。

三、存贮器结构及其寻址方式

1. 存贮器的组织

8088 具有二十位地址线，可以访问 1 兆字节的存贮空间。存贮器逻辑上被分成代码段、数据段、附加段和堆栈段，每个段的最大长度是 64K 字节。段起始地址由相应的段寄存器指出，其最后的四位规定为 0，即段寄存器组中存放着四个当前段的起始地址的高 16 位，指令利用某种寻址方式形成 16 位的逻辑地址，并将它用作为段内的偏移量，也就是说，将相应的段寄存器中的内容自动左移四位之后，与该偏移地址相加，就形成了二十位的物理地址，如图 12.5 所示。

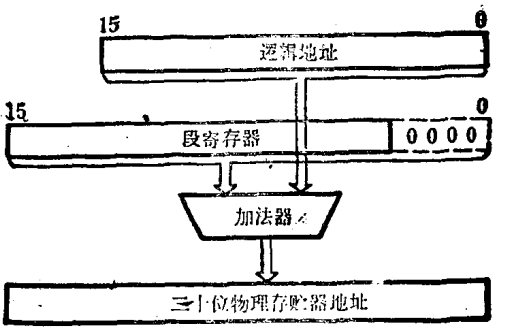


图 12.5　存贮器地址的形成

由于 8088 的外部接口采用的是 8 位的数据总线，存贮器也是按字节寻址的，因而存取字时必须连续访问两次存贮器，即先按指令中形成的地址存取第一个字节，然后地址加 1，存取下一个字节(高字节)。

图 12.6 给出了 8088 存贮器的组织情况，图 12.7 给出了存贮器中为系统所专用的一些存贮单元。

2．寻址方式

8088 的指令具有单字节和多字节多种形式，非常灵活，下面这种格式可被看成是一种典型的指令形式。

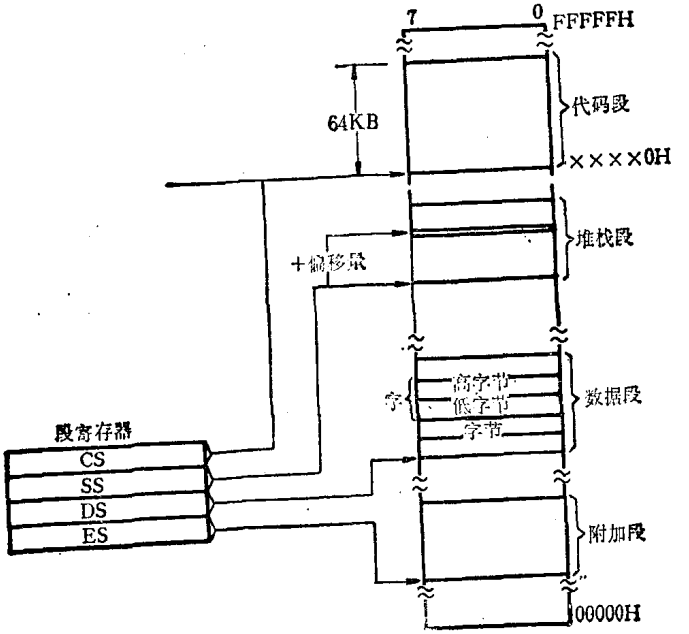


图 12.6　寄存器的组织

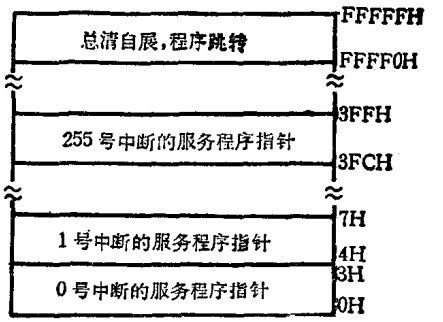
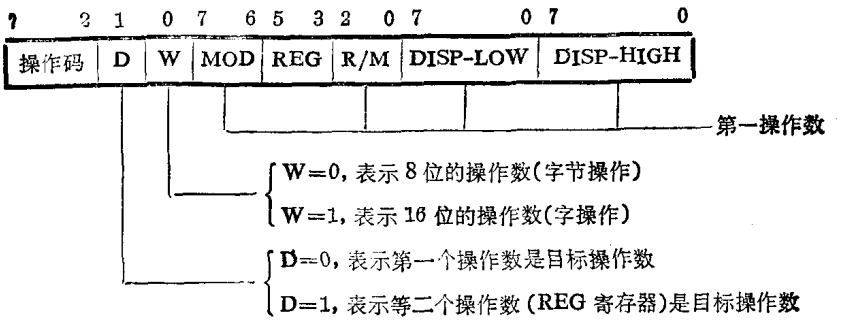


图 12.7　专用的存贮器单元



其中 REG 代表寄存器，其编码如下：

16 位(W = 1 时)　　8 位(W = 0 时)

000	AX	000	AL
001	CX	001	CL
010	DX	010	DL
011	BX	011	BL
100	SP	100	AH
101	BP	101	CH
110	SI	110	DH
111	DI	111	BH

8088 具有如下一些寻址方式：

1) 存贮器操作数

当 MOD = 00, 01, 10 时，说明第一操作数是存贮器操作数，其寻址方式由 MOD 字段和 R/M 字段联合确定。

直接寻址方式：此时 MOD = 00, R/M = 110，操作数的 16 位偏移地址放在指令码后面的两个字节当中。

间接寻址方式：8 位或 16 位带符号的编移量放在指令后面，操作数的实际地址还要通过

i) 基址寄存器

ii) 变址寄存器

iii) 基址寄存器与变址寄存器的和

才能获得。

2) 寄存器寻址

8088 的指令可以把某个通用寄存器、指示器或变址寄存器用作为操作数寄存器，当 MOD = 11 时，第一个操作数为寄存器，其操作数寄存器号由 R/M 字段指出。

3) 立即数

双操作数指令中有两个操作数，其中之一可以取立即数，立即数通常都是以 8 位或 16 位补码的形式放在指令当中的。

作为小结，表 12.1 列出了 8088 指令中第一操作数的各种寻址方式。

在 8088 指令的前面，可以加上段跨越前缀字节，其中的段寄存器的编码为：

00	ES
01	CS
10	SS
11	DS

表 12.2 对形成存贮器地址的各个分量进行了一个小结。8088 能够根据存贮器访问的类型自己选择某个段寄存器作为段基址，这种段寄存器被称为自动段基。假如在指令的前面加上段跨越前缀，则就可以把该前缀中给出的段寄存器用作访问存贮器的基地址，即用所谓的替换段基取代自动段基。然后把逻辑地址(有效地址)加到所选择的段寄存器上，就得到了真正的存贮器地址。

3. 对高级语言的支持

8088 的操作数寻址方法对高级语言的实现提供了有力的支持，使得程序员能够非常方便地实现对简单变量、数组、记录数组的访问，其实现方法如表 12.3 所示。

表 12.1 8088 的操作数寻址方式

MOD=00*: DISP=0
=01 : DISP=符号扩展到 16 位的单字节位移
=10 : DISP=(DISP-HIGH, DISP-LOW)

R/M=000: EA=(BX)+(SI)+DISP
=001: EA=(BX)+(DI)+DISP
=010: EA=(BP)+(SI)+DISP
=011: EA=(BP)+(DI)+DISP
=100: EA=(SI)+DISP
=101: EA=(DI)+DISP
=110: EA=(BP)+DISP*
=111: EA=(BX)+DISP

MOD=11:

R/M	8 位(W=0)	16 位(W=1)
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

* MOD=00 且 R/M=110 时，为直接寻址方式

表 12.2 8088 的各寻址分量

存贮器访问类型	自动段基	替换段基	段内地址(偏移)
取指令	CS	无	IP
堆栈操作	SS	无	SP
串指令源操作数	DS	CS、ES、SS	SI
串指令的目标	ES	无	DI
以 BP 为基址	SS	CS、ES、DS	EA
一般的数据读写	DS	CS、ES、SS	EA

表 12.3 几种数据结构的有效寻址

数据结构	数据存贮器		栈段
	没有基址	有基址	
简单变量	直接寻址	BX+偏移量	BP+偏移量
数组	SI DI	BX+SI BX+DI	BP+SI BP+DI
记录数组	SI+偏移量 DI+偏移量	BX+SI+偏移量 BX+DI+偏移量	BP+SI+偏移量 BP+DI+偏移量

当采用包含 BP 基址寄存器的寻址方式时，所访问的不是数据段中的数据，而是栈段中的数据。大家知道，在递归过程和块结构的程序设计语言当中，经常需要将数据存放在存贮器堆栈当中。

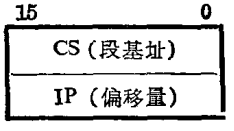
四、输入/输出

在8088当中，I/O操作最多可以寻址64K的I/O空间。当8088执行I/O操作时，地址线A_{19}～A_{16}为0，而用总线A_{15}～A_0来寻址I/O空间，显然，I/O操作的寻址范围可达64K。8088的I/O指令也分为固定转接口和可变转接口两种类型。对于固定转接口的I/O指令(直接I/O指令)而言，操作码后面的那个存贮器字节中存放着转接口的号码，故这类指令可以对256个I/O转接口(0～255)进行操作。而可变转接口的I/O指令则用DX寄存器作为指示器，即把DX的内容用作为16位的I/O地址，这样，就可以对整个I/O空间实现访问。

8088中既有8位的I/O指令，也有16位的I/O指令，但在实际上，8088的16位的输入输出指令都被转换成连续的两次8位操作，即一次字输入/输出操作将由两个字节输入/输出操作实现。I/O转接口的地址确定方法是与存贮器单元的定位方法相同的。

五、中断机构

8088的中断是由外部设备、中断指令或CPU本身所引起的，中断使得控制转移到程序中的一个新的位置。我们也可以把8088的中断操作分为软件中断和硬件中断两类。中断向量表位于内存的000～3FF这1K字节的空间中，每个中断向量占四个存贮器字节，它们分别指向各自的中断服务程序的入口地址，中断向量的格式是这样的：



在发生中断时，8088首先就将保护好当前CS和IP中的值，然后，将相应的中断向量装入CS及IP之中，转移到中断服务程序。中断向量表中可以包含多达256个对应于不同中断类型的中断向量。

对于系统中的每种中断，通常都给它一个特定的类型码，这些类型码实际上就是各中断向量在中断向量表中的序号，即类型码乘4就将得到中断向量的地址，因此，CPU通过中断的类型码就能够找到相应的中断向量．在发生外部中断时，外部中断系统(如8259A可编程中断控制器)就将在中断识别周期内，向CPU提供一个字节的中断类型码。不可屏蔽中断NMI的类型码被规定为2，至于内部中断的类型码则是由中断指令所指明的。

8088微处理器中有一个单独的中断引脚，即不可屏蔽中断(NMI)，它要比可屏蔽的中断(INTR)具有更高的优先级别。电源失效也许可以算是NMI的一个最为典型的例子。这种类型的中断不可以被屏蔽，故只要出现这类中断，CPU就应该响应它。但是对INTR来讲，只要把允许中断标志位清0，那么就相当于屏蔽掉了所有的INTR中断。此外，在中断响应期间不应该允许新的中断出现，故在响应任何中断(INTR、NMI、软件中断或单步)时，都要把中断允许标志位清0，这时无需关心原标志寄存器的状态，因为在对中断进行处理之前，原标志寄存器已经被压入存贮器堆栈了。除非用一指令来设置中断允许标志位，否则，在原来的标志寄存器被恢复之前，中断允许标志位总保持为0。8088的中断服务过程类似于中断的一般处理方式：保护现场、获得中断向量的地址、执行中断服务程序、恢复现场。

六、对实现多处理机系统的支持

8088为实现多处理机系统提供了有效的硬件支持．例如，8088的WAIT指令可用来实现8088与其它处理器(如8087)的同步，为多个处理器正确地并行工作提供了保障；HOLD和HLDA(或RQ/GT_0和RQ/GT_1)为多个总线主设备公用总线提供了方便的控制手段；LOCK指

令前缀又使得指令在执行期间，可以封锁总线，这就使得在多处理机系统中，对共享资源的访问变得相当方便。

七、8088 的指令系统

8088 的指令系统与 8086 完全相同，由于在上册当中已经非常详细地对 8086 的指令系统进行了介绍，故这里不再重复。

§12.1.3 8088 与 8086 的比较

8088 CPU 是一个 8 位处理器，它具有 8086 的内部结构，它的大多数内部功能都相同于 8086 的相应功能，这些从上面的介绍中就可以看出来。8088 管理外部总线的方法也类似于 8086，只是它每次只能处理 8 位，即存取 16 位的操作数时需要两个连续的总线周期。在软件工程师(程序员)看来，8088 除了在运行时间上与 8086 有所区别之外，其它都是相同的。它们具有相同的内部寄存器结构，具有相同的指令系统，所有指令在这两种处理器上产生的最终结果都是一样的。下面，主要讨论一下两者之间的区别。

从内部结构看，8088 与 8086 之间有三点不同，实质上，8088 对 8086 所作的这几点修改都是与其采用了 8 位数据接口总线这一事实直接相关的。

1．8088 中指令队列的长度是四个字节，而在 8086 当中，指令队列的长度是六个字节(或三个字)。缩短队列长度的目的在于免得总线接口部件 BIU 过多地使用总线去预取指令，这主要是因为考虑到在每次取 8 位的条件下，取指令需花去更多的时间，即需要更多的总线周期，假如 BIU 过分频繁地执行预取指令操作，那就有可能造成总线负载过重的现象。

2．为了进一步优化对指令队列的使用，8088 对预取指令所用的算法也适当地做了些更改，修改后的算法是：只要指令队列中有一个字节的自由空间（可供使用），8088 的 BIU 就将取出一个新的指令字节，装到指令队列中去。而 8086 则直要等到指令队列中有起码两个可用的字节空间时，才预取后继指令。

3．指令的内部执行时间也要受到 8 位数据总线接口的影响，所有 16 位的存贮器读/写操作都需要两个总线读/写周期，当然，CPU 还要受到取指令速度的限制。后面这种情况(取指令速度的影响)只有在 8088 执行一串极简单的指令时才会发生，而当使用较为复杂的 8088 指令时，8088 就有时间填满指令队列，这时，8088 的运行速度就会达到运行单元的极限速度。

由于 8088 与 8086 具有相同的执行部件，因而它们的软件是完全兼容的。也就是说，在 8086 上运行的所有程序可以不加任何修改，而由 8088 来执行。反之亦然。

8086 与 8088 之间的主要差别反映在它们的硬件接口上，尽管两者的引脚分布基本上相同，但其功能有这样几点区别：

1．在 8088 中，A_8～A_{15} 这八条引脚仅仅作为地址输出信号，而不再是与数据总线分时公用。对这些地址线的处理方式类似于 8085 中的高位地址线，即它们由内部锁存，且在整个总线周期内都保持有效。

2．对 8088 来讲，$\overline{\text{BHE}}$ 信号没有意义，因而将该信号删去了。

3．在 8088 处于最小模式工作方式时，$\overline{\text{SSO}}$ 提供了 $\overline{\text{SO}}$ 的状态信息，此输出信号仅仅在最小模式中的 34 号引脚上出现。DT/$\overline{\text{R}}$、IO/$\overline{\text{M}}$ 和 $\overline{\text{SSO}}$ 提供了最小模式下完整的总线状态信息。

4．IO/$\overline{M}$ 已被变得与 MCS-85 总线结构兼容，而不再是 8086 中的 M/$\overline{IO}$。

5．为了能够用 ALE 锁存状态信息，在进入 HALT 时，最小模式下的 ALE 信号被推迟了一个时间周期。

§12.2 8284A 时钟发生和驱动器

§12.2.1 概 述

8284A 是一个时钟发生器/驱动器芯片，它为 iAPX86、88 微处理器及其它外设芯片提供所需要的时钟信号，在 8086 系列芯片所组成的系统当中，8284A 可以作为整个系统工作的“心脏”，心脏停止跳动，整个系统当然也就停止活动了。事实上，系统中的时钟信号也确实是不可缺少的，这也是我们介绍 8284A 时钟发生/驱动器的理由所在。除了为系统提供时钟信号之外，8284A 中还具有系统复位控制逻辑，还能够提供起同步作用的 READY（准备好）信号。8284A 具有如下特色：

1．能够产生 iAPX86、88 微处理器所需的系统时钟脉冲，它所提供的时钟频率有 5 MHz 和 8 MHz 两种（8284A-1 能够产生时钟频率为 10 MHz 的时钟信号）；

2．采用晶体或 TTL 信号作为频率源；

3．提供了一定的同步机构（READY 信号）；

4．采用十八只引脚的封装；

5．单一的 +5V 电源；

6．能够产生系统复位输出信号；

7．能与其它的 8284A 时钟保持同步。

§12.2.2 8284A 的引脚结构

图 12.8 给出了 8284A 的内部结构框图，图 12.9 则给出了 8284A 的引脚结构。

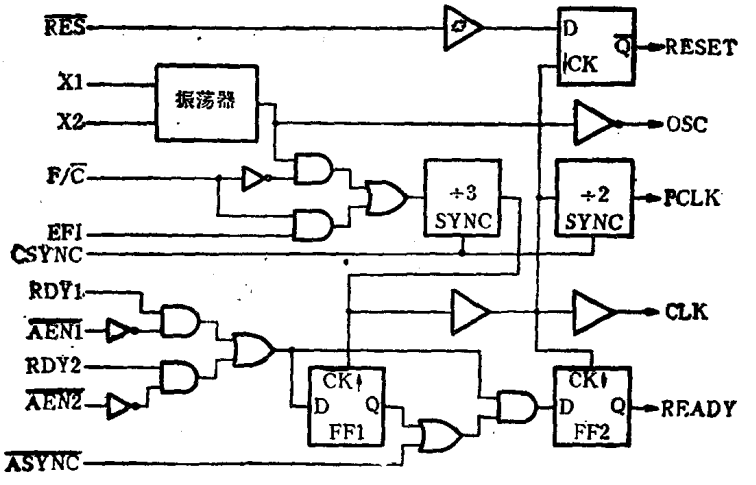


图 12.8 8284A 的结构框图

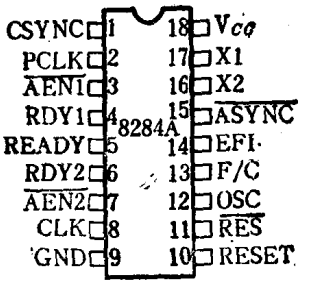


图 12.9 8284A 的引脚结构

下面，简单地介绍一下 8284A 的十八个引脚及其相应的功能。

$\overline{AEN1}$、$\overline{AEN2}$ 输入信号

这是两个低电平为有效的信号，它们分别用来制约相应的总线准备好信号 RDY1 或 RDY2，$\overline{AEN1}$ 使 RDY1 信号生效，而 $\overline{AEN2}$ 则使 RDY2 信号生效。在允许处理器访问两根多主设备总线的系统中，这两个 AEN 是非常有用的。在非多总线主系统结构当中，$\overline{AEN1}$ 和 $\overline{AEN2}$ 信号的输入端均接低电平。

RDY1、RDY2 输入信号

这是两个高电平为有效的信号，表示总线准备好。当 RDY 为高电平时，则说明发生该信号的设备（挂在系统数据总线上）已经收到数据，或表示该设备中的数据可供系统使用。正象上面指出的那样，RDY1 受 $\overline{AEN1}$ 制约，RDY2 受 $\overline{AEN2}$ 制约。

$\overline{ASYNC}$ 输入信号

该信号所起的作用就是选择 READY（准备好）的同步方式，即该输入信号规定了 READY 逻辑的同步方式。具体讲就是，当 $\overline{ASYNC}$ 为低电平时，则提供两级 READY 同步；而当 $\overline{ASYNC}$ 为高电平时，则提供一级 READY 同步。这从图 12.8 中可以很清楚地看出来。

READY 输出信号

准备好信号，这是一个高电平为有效的信号，它是由输入信号 RDY 经同步之后而形成的。由于 RDY 相对于处理器时钟来讲是异步产生的，因此，在把它送给处理器之前应该使其与时钟同步。在确保处理器已经收到 READY 信号一段时间之后，就将清除该 READY 信号。

X1、X2 输入信号

这是与晶体相连的两个输入端。其中，晶体的振荡频率是处理器所需时钟频率的三倍。

F/$\overline{C}$ 输入信号

频率/晶体选择信号，8284A 利用该输入信号来选择其工作方式。当 F/$\overline{C}$ 为低电平信号时，则由晶体产生处理器的时钟信号；而当 F/$\overline{C}$ 为高电平时，则时钟信号 CLK 则是由外部频率源（EFI）所导出的。

EFI 输入信号

这是一个外部频率输入端。当 F/$\overline{C}$ 与高电平信号相接时，那么，处理器时钟 CLK 则由 EFI 所送入的频率产生。其输入为方波信号，频率是 CLK 输出信号所需频率的三倍。

CLK 输出信号

处理器时钟信号。该时钟输出信号由处理器以及其它直接与处理器局部总线相连的设备所公用，它也是整个系统工作的基准。CLK 的频率是晶体振荡频率或 EFI 输入频率的三分之一。CLK 的输出高电平为 4.5V，以驱动 MOS 器件。

PCLK 输出信号

PCLK 是供外设用的又一个时钟输出信号，其输出频率是 CLK 的一半。

OSC 输出信号

OSC 是振荡器的输出端，即是内部振荡电路的 TTL 级输出信号，其频率就等于晶体的振荡频率。

$\overline{RES}$ 输入信号

这是一个复位信号输入端。$\overline{RES}$ 是一个低电平为有效的信号，它被用来产生 RESET（复

位)信号。8284A 提供了一个施密特触发器,以使它能够产生长短适当的复位信号。

RESET 输出信号

复位信号。RESET 是一个高电平有效的输出信号,它被用来复位 8086 系列的处理器。该信号的时序特征是由 $\overline{RES}$ 信号所确定的。

CSYNC 输入信号

CSYNC 也是一个高电平为有效的信号,该信号使得多个 8284A 能够同步地提供时钟信号。当 CSYNC 为高电平时,内部的一些计数器就将被复位;而当 CSYNC 变为低电平时,又将使得这些内部计数器恢复计数。CSYNC 需要与外部的 EFI 信号保持同步,但在使用内部的晶体振荡器时,CSYNC 应该接地。

GND 接地

V_{CC} +5V 电源

§12.2.3 8284A 的功能概述

由图 12.8 可以看出,8284A 由一个晶体振荡器、一个三分频计数器、多总线准备好(READY)信号控制逻辑以及复位(RESET)信号产生逻辑这几个部分组成。下面,我们准备分别对这几个部分做些讨论。

一、晶体振荡器

设计 8284A 振荡器电路的主要目的就是希望能由此导出基本的系统工作频率。在8284A 当中,晶体的振荡频率应为 CPU 所需时钟频率的三倍。前面我们已经讲过,X1 和 X2 是与晶体相连的两个输入端,振荡器的输出信号经过缓冲并送到 OSC 输出端,这样,其它的系统定时信号就能够从该稳定的、晶体控制的振荡源中导出。

二、时钟发生器

时钟发生器由一个三分频的同步计数器及一个起禁止计数作用的输入端所组成,其中的清除输入端(CSYNC)使得输出时钟能够与某个外部事件(如另一个 8284A 时钟)同步。对 8284A 时钟发生/驱动器而言,必须使 CSYNC 与 EFI 保持同步,这是通过两个肖特基触发器实现的,如图 12.10 所示。

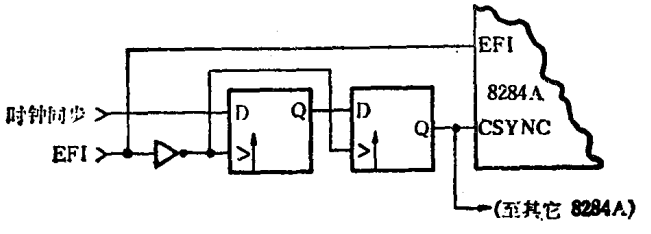


图 12.10 CSYNC 的同步作用

F/$\overline{C}$ 输入信号是用来选择三分频计数器的信号源的,即根据 F/$\overline{C}$ 的取值情况,决定是把晶体振荡器选为三分频计数器的时钟输入信号,还是把 EFI 作为三分频计数器的时钟,如图 12.11 和图 12.12 所示。

倘若把 EFI 输入信号用作时钟的信号源,那么,其晶体振荡器部分就能够用作另一个时钟源,由 OSC 端得到振荡器的输出信号。

CLK 输出端是一个 MOS 型时钟驱动器,它可以直接驱动 iAPX 86、88 处理器。PCLK 输出的是 TTL 级的外设时钟信号,其输出信号的频率是 CLK 的二分之一。

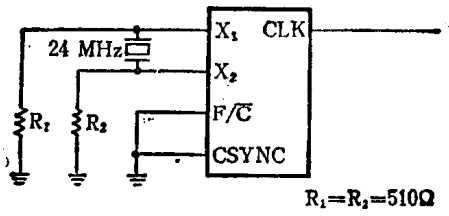


图 12.11 晶体振荡器作为信号源(用 X1、X2)

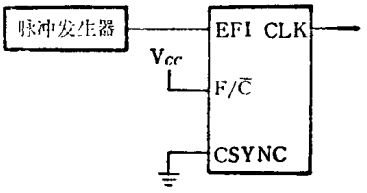


图 12.12 外部频率作为信号源(用 EFI)

三、复位逻辑

复位逻辑产生复位定时信号，它由一个施密特触发器和一个同步触发器组成。复位信号是与 CLK 的下降沿同步的。通过利用 8284A 的这一功能，利用一个简单的 RC 网络就能够产生合上电源时的复位信号。

四、READY 信号同步机构

8284A 提供了两个 READY (准备好)输入端，即 RDY1 和 RDY2，以适应两个多主设备的系统总线的需要。RDY1 和 RDY2 均有自己的一个制约信号，其制约信号分别为 $\overline{AEN1}$ 和 $\overline{AEN2}$，制约信号使得相应的 RDY 信号生效。假如使用的不是一个多主设备的系统，那么，就应把 $\overline{AEN}$ 引脚接地。

由于 RDY1 和 RDY2 相对于处理器时钟来讲是异步产生的，因此，在把它们送到处理器之前，为了保证满足一定的建立和保持时间需要，就对 READY 信号提出了同步要求。$\overline{ASYNC}$ 输入信号确定了 READY 同步操作的两种方式。当 $\overline{ASYNC}$ 为低电平时，则表示为准备好输入信号提供了两级同步，异步发出的准备好输入信号(RDY)首先在触发器 1 处与 CLK 的上升沿获得首次同步，然后又在触发器 2 处与下一个 CLK 的下降沿再次获得同步，在这之后，READY 输出信号将变成高电平。至于准备好输入信号 (RDY) 由高电平变为低电平时(有效变为无效)，它就将直接在触发器 2 处与 CLK 的下降沿取得同步，并将 READY 输出信号置为无效状态。

当 $\overline{ASYNC}$ 为高电平时，READY 同步逻辑中的第一个触发器不再起作用。RDY 输入信号将直接在触发器 2 处与 CLK 的下降沿取得同步，然后，向处理器发出 READY 信号。究竟采用哪种 READY 同步方式，主要取决于 RDY 所需要的建立时间。若时间允许的话，可以采用后一种 READY 同步方式，否则，就应该用前一种 READY 同步方式，以便系统中的那些异步工作设备能够正确地执行任务。

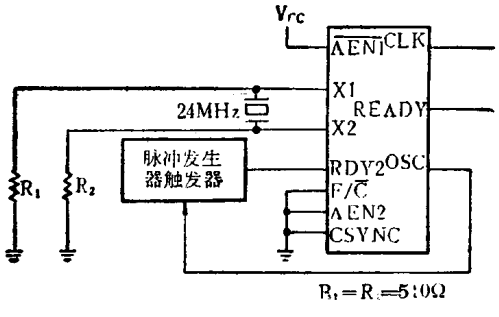


图 12.13 时钟准备好(用 X1、X2)

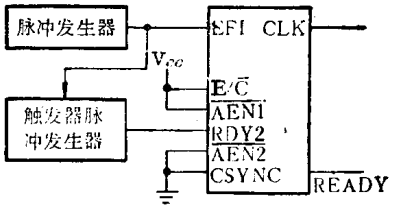


图 12.14 时钟准备好(用 EFI)

$\overline{\text{ASYNC}}$ 在每个总线周期内都可以发生变化，这就使我们能够为系统中的各个设备选择适当的同步方式。

图 12.13 和图 12.14 给出了用于指出时钟准备好的两种结构形成。

§12.3 8288 总线控制器

§12.3.1 概　　述

8288 总线控制器是一种二十根引脚的双极型元件，它可以用于中大规模的 iAPX 86、88 系统当中。8288 总线控制器除了具有一定的总线驱动能力之外，还能够提供一些命令和控制功能，利用 8288 可以在一定程度上优化整个系统的性能。8288 既适用于多总线主设备的系统总线，也适用于独立的 I/O 总线。我们可以把 8288 的特性归纳为这样几点：

1. 双极型的驱动能力；
2. 提供了一些超前命令；
3. 使系统的结构更为灵活；
4. 三态的命令输出驱动器；
5. 可用于 I/O 总线；
6. 它使得跟一个或两个多总线主设备的总线的接口更为简单。

§12.3.2 8288 的引脚结构

图 12.15 和图 12.16 分别表示 8288 总线控制器的结构框图及其引脚结构。本小节将着重介绍一下 8288 的二十根引脚以及相应的功能。

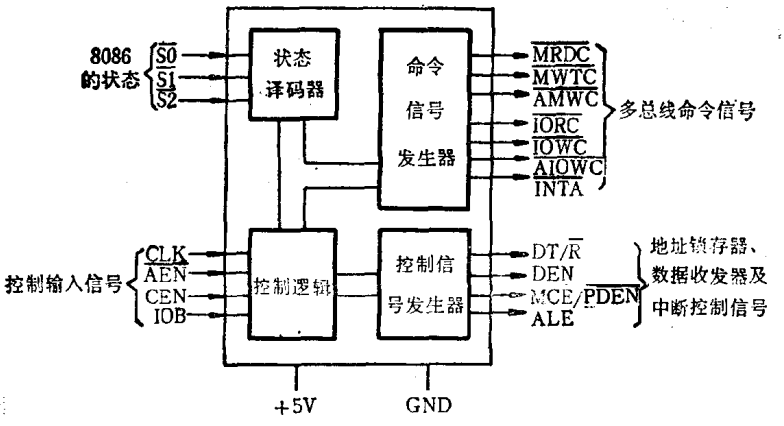


图 12.15　8288 的结构框图

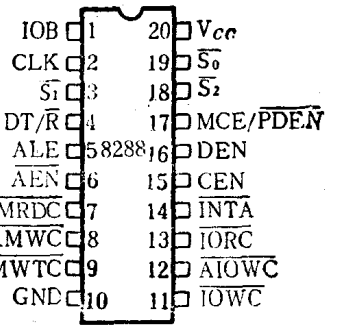


图 12.16　8288 的引脚结构

$\overline{S_0}$、$\overline{S_1}$、$\overline{S_2}$　输入信号

这是三个状态输入端，其输入信息来自 8086、8088 或 8089 处理器。8288 对这些状态信号进行译码，并在适当的时候产生相应的命令或控制信号。当没用到这些信号时，它们将全处于高电平状态。

CLK 输入信号

时钟输入端. CLK 是 8284 时钟发生器所发出的时钟信号，8288 利用它来确定何时发出命令与控制信号.

ALE 输出信号

允许地址锁存信号. 该信号所起的作用就是打通地址到地址锁存器的通路，即选通地址，并将它送至地址锁存器中锁存起来. ALE 是一个高电平为有效的信号，地址锁存过程发生在 ALE 信号的下降沿.

DEN 输出信号

DEN 也是一个高电平为有效的信号，该信号用来把数据收发器挂到局部数据总线或系统数据总线上.

DT/$\overline{R}$ 输出信号

发送/接收数据信号. DT/$\overline{R}$ 确定数据流经数据收发器的方向. DT/$\overline{R}$ 为高电平时，表示发送数据，即执行写 I/O 或写存贮器操作；而当 DT/$\overline{R}$ 为低电平时，则表示接收数据，即执行的是读 I/O 或读存贮器命令.

$\overline{AEN}$ 输入信号

这是一个低电平为有效的信号，在 $\overline{AEN}$ 变为低电平（有效）至少 115 ns 之后，8288 总线控制器才可以发出命令. 一旦 $\overline{AEN}$ 变为高电平，命令输出驱动器就将进入高阻抗状态. 但在 8288 处于 I/O 总线工作模式时(IOB 为高电平)，$\overline{AEN}$ 就没有上述作用，即它不会影响 I/O 命令的发出.

CEN 输入信号

命令允许信号. 当此信号为低电平时，8288 的所有命令输出端和 DEN 及 $\overline{PDEN}$ 控制输出端均被置为无效状态，只有当 CEN 为高电平时，上述这些输出端才能发出各自的命令或控制信号.

IOB 输入信号

输入/输出总线工作模式. 当 IOB 端输入的是高电平时，8288 就以 I/O 总线模式工作；而当它接至低电平时，8288 则以系统总线模式工作.

$\overline{AIOWC}$ 输出信号

超前的写 I/O 命令. 为了提早通知 I/O 设备它需执行一个写命令，$\overline{AIOWC}$ 将在机器周期开始之前发出一个写 I/O 命令，其时序与读命令相同. 这是一个低电平为有效的信号.

$\overline{IOWC}$ 输出信号

I/O 写命令. $\overline{IOWC}$ 是一个低电平为有效的信号，该命令将使某个 I/O 设备读进数据总线上的数据.

$\overline{IORC}$ 输出信号

I/O 读命令. $\overline{IORC}$ 是一个低电平为有效的信号，它命令某个 I/O 设备将其中的数据送到数据总线上.

$\overline{AMWC}$ 输出信号

超前的存贮器写命令. 为了提前通知存贮器设备它要执行一个写命令，则由 $\overline{AMWC}$ 在此机器周期开始之前发出一个存贮器写命令. 其时序与读命令相同. $\overline{AMWC}$ 是低电平有效的一个输出信号.

$\overline{\text{MWTC}}$ 输出信号

写存贮器命令。$\overline{\text{MWTC}}$ 命令存贮器把数据总线上的数据记录下来，它是一个低电平有效的信号。

$\overline{\text{MRDC}}$ 输出信号

读存贮器命令。$\overline{\text{MRDC}}$ 命令存贮器把其中的数据送至数据总线上。它也是一个低电平为有效的信号。

$\overline{\text{INTA}}$ 输出信号

$\overline{\text{INTA}}$ 为中断响应信号，该命令通知发出中断申请的设备，其中断请求已经得到响应，这时，此设备应该把中断向量地址等有关信息送到数据总线。$\overline{\text{INTA}}$ 也是一个低电平为有效的信号。

MCE/$\overline{\text{PDEN}}$ 输出信号

该引脚具有双重功能，即当 MCE 用时和当 $\overline{\text{PDEN}}$ 用时的功能是不一样的。

当 IOB 接低电平信号时，该引脚用作允许主设备级联 (MCE) 信号，在中断序列中，MCE 信号就将出现，它被用来从某个主 PIC（优先级中断控制器）中读出级联地址，并将所读出的地址送到数据总线。MCE 是高电平为有效的一个信号。

当 IOB 接高电平信号时，该引脚就代表允许外设数据 ($\overline{\text{PDEN}}$) 信号。$\overline{\text{PDEN}}$ 相当于把数据总线收发器挂到 I/O 总线上，可见，$\overline{\text{PDEN}}$ 对 I/O 总线的影响与 DEN 对系统总线的作用是相同的。$\overline{\text{PDEN}}$ 是一个低电平为有效的信号。

V_{cc} +5V电源。

GND 接地线。

§12.3.3 8288的功能

我们知道，8288 是通过对 8086、8088 或 8089 的三根状态线 ($\overline{S_0}$, $\overline{S_1}$, $\overline{S_2}$) 进行译码，从而确定它该发出什么命令的。下表列出了各种状态组合所代表的意义。

表 12.3 状态 ($\overline{S_0}$, $\overline{S_1}$, $\overline{S_2}$) 表

$\overline{S_2}$ $\overline{S_1}$ $\overline{S_0}$	处理器状态	8288 的命令
0 0 0	中断响应	$\overline{\text{INTA}}$
0 0 1	读 I/O 转接口	$\overline{\text{IORC}}$
0 1 0	写 I/O 转接口	$\overline{\text{IOWC}}$, $\overline{\text{AIOWC}}$
0 1 1	停机	无
1 0 0	访问代码	$\overline{\text{MRDC}}$
1 0 1	读存贮器	$\overline{\text{MRDC}}$
1 1 0	写存贮器	$\overline{\text{MWTC}}$, $\overline{\text{AMWC}}$
1 1 1	无源	无

根据 8288 总线控制器所处的工作模式，其命令是以两种不同的方式发出的。

如果 IOB 引脚与高电平信号相接，那么，我们就称 8288 处于 I/O 总线工作模式。在 I/O 总线工作模式中，所有的 I/O 命令线 ($\overline{\text{IORC}}$、$\overline{\text{IOWC}}$、$\overline{\text{AIOWC}}$、$\overline{\text{INTA}}$) 都可以起作用，而与 AEN 的状态无关。一旦处理器启动某个 I/O 命令，8288 就利用 $\overline{\text{PDEN}}$ 和 DT/$\overline{\text{R}}$ 激活相应的

命令线，以控制 I/O 总线收发器。在这种结构方式当中，由于没有提供总线裁决机构，因此不能把 I/O 命令用来控制系统总线。但这种工作模式却使 8288 总线控制器能去管理两组外部总线。当 CPU 希望访问 I/O 总线时，无需进行等待，但对一般的存贮器访问而言，在开始访问之前，通常总需要一个“总线准备好”信号($\overline{AEN}$ 为低电平)。假如在某个多处理器系统当中，存在一些专供一个处理器使用的 I/O 设备的话，那么，采用这种工作模式还是有利可图的。

倘若 IOB 引脚所连接的是低电平信号，则称 8288 处于系统总线工作模式之中。在这种模式中，至少要等到信号 $\overline{AEN}$ 为低电平(有效)的时间超过 115 ns 之后，8288 才能发出命令。当总线可供使用时，总线裁决机构将通过 $\overline{AEN}$ 信号线通知总线控制器，告诉它总线处于空闲状态。在这种模式下，无论是存贮器命令，还是 I/O 命令，它们都要经过总线裁决机构认可后才能生效。在系统中只有一条总线时，应采用这种工作模式，这里的 I/O 设备和存贮器都是由多个处理器所共享的。

由于 8288 提供了超前写命令，因而使之可以在写周期开始之前就启动写过程，这就能够在一定程度上避免处理器进入没必要的等待状态。8288 有以下七个命令输出端：

$\overline{MRDC}$：存贮器读命令

$\overline{MWTC}$：存贮器写命令

$\overline{IORC}$：I/O 读命令

$\overline{IOWC}$：I/O 写命令

$\overline{AMWC}$：超前的存贮器写命令

$\overline{AIOWC}$：超前的 I/O 写命令

$\overline{INTA}$：中断响应

其中，$\overline{INTA}$ (中断响应)在中断周期中的作用和 I/O 读命令的作用类似，其目的就是想通知中断源(发出该中断请求的设备)，本中断请求已经得到响应，要求此设备把有关信息送到数据总线上，以便对中断进行处理。

除了上面所介绍的命令输出端之外，8288 还有一些控制输出信号，它们是 DEN、DT/$\overline{R}$、ALE 和 MCE/$\overline{PDEN}$。其中，DEN 确定何时使得外部总线与局部总线相通，DT/$\overline{R}$ 则确定了数据传送的方向，这两个信号通常是接到收发器的片选端和方向控制端。ALE 则用来把当前地址送到地址锁存器中去，它在每个机器周期内都将出现，在欲进入暂停状态时，也可以把它用来锁存状态信息($\overline{S_0}$, $\overline{S_1}$, $\overline{S_2}$)。MCE/$\overline{PDEN}$ 引脚的功能是随着 8288 工作模式的改变而改变的，当 8288 处于 I/O 总线工作模式时 (IOB 接高电平)，$\overline{PDEN}$ 起作用，它被用作 I/O 或外设系统总线的数据允许信号，若 8288 工作在系统总线模式之中 (IOB 接低电平)，那么，在中断响应周期内就将使用 MCE 信号，在对任何中断的处理过程中，总存在着两个中断响应周期，在第一个中断响应周期内，不会发生任何数据或地址传送操作，在此期间，MCE 信号被屏蔽掉了。而在第二个中断响应周期刚要开始之前，MCE 信号开始生效，并把作为主设备的某个 PIC 的级联地址送到处理器的局部总线上，然后，在 ALE 的控制下将该地址送至地址锁存器。在第二个中断响应周期的前沿，被寻址的 PIC 把中断向量送到系统数据总线，这样，处理器就能够取得中断向量信息，实现对本次中断的处理。假如系统中仅包含了一个 PIC，那么就没必要使用 MCE 信号，在这种情况下，第二个中断响应信号就会将中断向量送至处理器总线。

§12.4 8289 总线裁决器

§12.4.1 概　　述

近年来，随着微型机性能的不断提高及价格的迅速下降，人们对多处理器系统的设计越来越感兴趣，多机系统、分布处理是当今计算机界所研究的热门课题，可以断言，下一代计算机的体系结构肯定是以多机系统为基础的，是分布式的处理系统．当然，为了设计多机系统，还有许多问题有待解决，多机系统中的硬件、软件、拓扑结构及通信体系都还值得研究．由微处理器组成的分布处理系统具有许多独特的优点，这主要表现在经济性、灵活性、可靠性、高效性这几个方面，特别是16位微型机的出现，又增加了方便性这一优点，即用16位微处理器可以很方便地组成分布处理系统，这是因为它们都为构造多机系统提供了有力的硬件、软件支持，从前面我们对8086/8088的介绍中可以看出这一特色．但所谓方便性，也并非说无需做任何工作，在多处理器系统的设计过程中还是有一些问题需要解决的，比如，如何保证系统中的各处理器同步地进行工作，避免或解决对共享资源的竞争问题等等，8289总线裁决器正是为了解决8086系列多处理器系统中的总线共享问题而设计的．

8289总线裁决器与8288总线控制器一道工作，以实现多个8086系列的处理器（8086、8088、8089）与系统总线之间的接口，对于处理器本身来讲，它并不知道总线裁决器的存在，而是认为系统总线是为它所独占的，这就为使用处理器提供了极大的方便，至于各处理器能否获得总线，何时获得总线这样一些问题均由8289总线裁决器解决．

8289总线裁决器是一个二十引脚的双极型元件，它主要用在中大规模的8086/8088系列多主设备/多重处理系统当中，为多个总线主设备系统提供了总线裁决机构．我们可以把8289的特性归纳成下面几条：

1．提供了多个总线主设备系统中系统总线的使用协议；
2．使作为总线主设备的诸处理器与多个主设备的总线同步地进行工作；
3．与8288总线控制器的接口很简单；
4．它具有四种工作方式，为组成灵活的系统提供了保障；
5．与MULTIBUS标准总线兼容；
6．为工作在远程模式下的8089 I/O处理器提供了系统总线裁决机构；
7．单一的 +5V 电源．

§12.4.2 8289的引脚结构与功能

图12.17和图12.18分别给出了8289总线裁决器的结构框图及引脚结构．本小节将重点介绍8289的诸引脚及相应的功能．

$\overline{S0}$、$\overline{S1}$、$\overline{S2}$ 输入信号

这是由8086、8088或8089处理器发来的三个状态输入端，8289通过对这三个状态信号进行译码，从而开始总线请求或释放操作．（见表12.4）．

CLK 输入信号

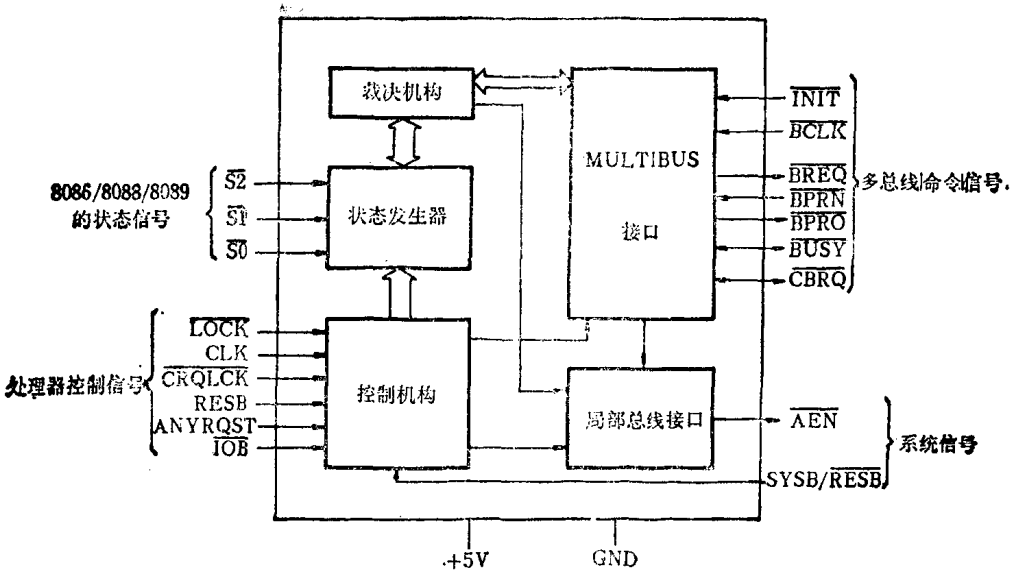


图 12.17　8289 的结构框图

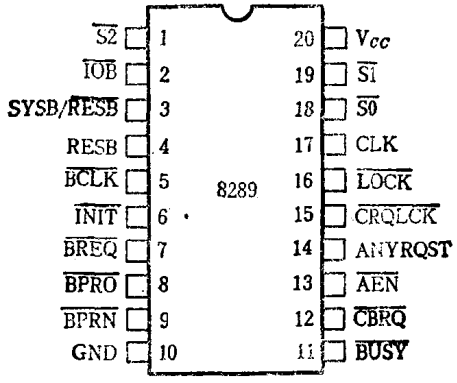


图 12.18　8289 的引脚结构

时钟输入端。CLK 信号来自 8284 时钟发生/驱动器，它用来确定总线裁决器何时开始工作。

LOCK　输入信号

这是一个总线封锁信号。当某个处理器决定不允许其裁决器把系统总线让给任何别的总线裁决器时，它就将 LOCK 信号置为低电平，即发出总线封锁信号，这时，不管其它总线裁决器的优先级有多高，它们都不能取得系统总线(注意，我们这里的系统总线一般都有多个总线主)。

CRQLCK　输入信号

这是一个公共请求封锁信号，当 CRQLCK 有效（低电平）时，其裁决器就不会把系统总线让给任何其它通过 CBRQ 请求总线的总线裁决器。

RESB　输入信号

该信号通知总线裁决器，它将在具有系统总线和一根驻留总线的系统中工作。当 RESB 接高电平信号时，多主设备的系统总线是作为 SYSB/RESB 输入信号的函数而被请求和释放的；而当它接低电平信号时，SYSB/RESB 输入信号不再起作用。

ANYRQST 输入信号

该信号使得可以把系统总线让给优先级较低的总线仲裁器，就好象这些裁决器具有更高的优先权一样，也就是说，当某个优先级别较低的总线裁决器请求使用总线时，总线将尽可能快地被释放，以使低优先权的裁决器也可以取得总线。具体讲，当 ANYRQST 接低电平信号时，总线将根据表 12.4 的规定被释放；倘若 ANYRQST 接高电平且信号 $\overline{\text{CBRQ}}$ 有效时，总线就将在当前的总线周期结束时被释放，即低电平的 $\overline{\text{CBRQ}}$ 和高电平的 ANYRQST 结合起来，将迫使 8289 总线裁决器在当前传送周期之后释放系统总线。值得注意的是，在释放总线时，将把 $\overline{\text{BREQ}}$ 信号置为高电平。

$\overline{\text{IOB}}$ 输入信号

$\overline{\text{IOB}}$ 和上面介绍的两个引脚 RESB 和 ANYRQST 在给定的系统当中通常都是固定的，即一般来讲，它们不随时间变化。$\overline{\text{IOB}}$ 信号所起的作用就是让 8289 工作在具有 I/O 总线和多主设备的系统总线环境之中。在这种系统当中，总线裁决器将根据 $\overline{S_2}$ 的状态确定它是该请求使用系统总线，还是该释放多主设备的系统总线。在处理器执行 I/O 命令的过程中，它就允许释放系统总线；而当处理器执行存贮器命令时，那它就要请求使用系统总线。中断周期都被当成 I/O 命令来处理，它们都被看作是来自外部总线 (I/O 总线)的。

$\overline{\text{AEN}}$ 输出信号

$\overline{\text{AEN}}$ 是 8289 送给处理器的地址锁存器的一个输出信号，该信号还可以送到 8288 总线控制器及 8284 A 时钟发生器。利用 $\overline{\text{AEN}}$ 可以把总线控制器和地址锁存器的输出置为三态。

SYSB/$\overline{\text{RESB}}$ 输入信号

SYSB/$\overline{\text{RESB}}$ 是给总线裁决器的一个输入信号，当总线裁决器处于 S. R. 工作模式(RESB 接高电平信号)时，该信号确定了系统总线何时被请求，以及何时允许释放系统总线。这个信号通常是由驻留地址总线上的某个译码器或 PROM 所产生的。当引脚 SYSB/$\overline{\text{RESB}}$ 的状态为高电平时，裁决器就将请求 S R. 模式中的系统总线；而当该引脚的状态为低电平时，其裁决器就允许系统总线被释放。

$\overline{\text{CBRQ}}$ 输入/输出信号

$\overline{\text{CBRQ}}$ 代表公共的总线请求。当作为输入信号使用时，该信号告诉此裁决器是否存在其它优先级较低的裁决器在请求使用系统总线。系统总线上所有 8289 总线裁决器的 $\overline{\text{CBRQ}}$ 引脚都被连到一起。

占有当前传送周期的总线裁决器自身不会把 $\overline{\text{CBRQ}}$ 置为低电平，但与该 $\overline{\text{CBRQ}}$ 引脚相连的任何其它裁决器却能够向系统总线发出使用总线的请求。一旦满足适当的释放条件，当前占有总线的裁决器就将降下其 $\overline{\text{BREQ}}$ 信号，并释放总线。当 $\overline{\text{CBRQ}}$ 为低电平且 ANYRQST 为高电平信号时，多主设备的系统总线在每个传送周期之后都将被释放。

$\overline{\text{INIT}}$ 输入信号

初始化信号，该信号被用来复位系统总线上的所有总线裁决器，在该初始化过程结束后，没有任何裁决器在占用系统总线。

$\overline{\text{BCLK}}$ 输入信号

$\overline{\text{BCLK}}$ 是多总线主设备的系统总线时钟信号，系统总线上的所有接口信号都与 $\overline{\text{BCLK}}$ 同步。

$\overline{\text{BREQ}}$ 输出信号

该信号代表总线请求，它是一个低电平有效的输出信号．在并行优先权判别方式中，总线裁决器通过激活其 $\overline{\text{BREQ}}$ 信号，来请求使用多个主设备的系统总线。

$\overline{\text{BPRN}}$ 输入信号

$\overline{\text{BPRN}}$ 是一个总线优先权输入端．当它为低电平（有效）时，其裁决器就知道它可以在下一个 BCLK 的下降沿获得系统总线．也就是说，$\overline{\text{BPRN}}$ 告诉裁决器，该裁决器是当前请求使用总线的裁决器中的优先权最高者．而当 $\overline{\text{BPRN}}$ 变为无效时，则说明该裁决器把优先占有总线的权利让给了某个优先级别更高的总线裁决器。

$\overline{\text{BPRO}}$ 输出信号

总线优先权输出端．在串行优先权分解方式中，$\overline{\text{BPRO}}$ 总是接到下一个优先级较低的裁决器的 $\overline{\text{BPRN}}$ 输入端，即把所有总线裁决器连成一个菊链。

$\overline{\text{BUSY}}$ 输入/输出信号

总线忙信号，$\overline{\text{BUSY}}$ 信号被用来通知总线上的所有裁决器，具有多个主设备的系统总线何时可供使用．当系统总线可供使用时（处于空闲状态），发出总线请求的裁决器中的优先级最高者（由 $\overline{\text{BPRN}}$ 确定）将取得总线，并将 $\overline{\text{BUSY}}$ 信号置为低电平，以阻止其它裁决器使用该总线．一旦此总线裁决器完成总线操作，它就允许 $\overline{\text{BUSY}}$ 信号变为高电平（不忙），从而也就使得另一个总线裁决器可以获得系统总线。

§12.4.3 8289 的工作过程

总线裁决器 8289 与总线控制器 8288 结合起来，为 8086 系列处理器与具有多个主设备的系统总线之间的接口提供了方便，对于系统中的各处理器来讲，它们无需知道总线裁决器的存在，它们都认为系统总线是为自己所独占的．但实际上，在处理器没有占用系统总线时，其裁决器就能够阻止其总线控制器 8288、数据收发器及地址锁存器去访问系统总线，即迫使所有的总线驱动器输出端进入高阻抗状态．由于此时该 8288 不会发出任何命令，那么，对该处理器来讲，系统总线呈“未准备好”状态，处理器也就进入等待状态，并将保持等待状态直到对应的总线裁决器获得了系统总线的使用权，而一旦取得系统总线，该裁决器也就将允许其总线控制器、数据收发器及地址锁存器访问系统总线，在处理器的数据传送命令经 8288 发出之后，被访问的从设备就将返回一个传送响应（XACK）信号给处理器，表示已准备好进行数据传送操作，然后，由处理器完成其传送周期．由此可见，总线裁决器的作用就是协调系统总线上各处理器（总线主设备）的动作，以避免多个总线主设备之间的竞争问题。

由于系统总线上可能有多个总线主设备，因而也就有可能出现几个主设备都在请求使用总线的情况，即发生总线争用的现象，为了解决这一问题，就必须建立一些规定，以确定占用总线先后次序．8289 对这一问题的解决方法是以优先权的概念为基础的．一般说来，当优先级别较低的某个主设备完成其当前的传送周期时，优先级别较高的那些主设备就将获得总线．但对于优先级别较低的那些总线主设备来讲，只有当优先级较高的主设备不在访问系统总线时，它们才能够获得总线．为了允许总线裁决器把总线让给某个优先级别更低的主设备，8289 提供了一个 ANYRQST 选择信号，它可以让裁决器在每个传送周期之后释放总线（当有使用总线的请求时），这就好象提高了请求总线的那些裁决器的优先级别．倘若系统中不存在其它

的总线主设备在请求总线，那么，只要占用总线的那个处理器没有进入暂停 (HALT) 状态，其裁决器就将保持总线的控制权。也就是说，裁决器本身不会自愿释放系统总线，总线所有权只有在其它主设备请求总线时才有可能被剥夺，当然，正象上面指出的那样，暂停状态是一种也是唯一的一种例外情况。8289 总线裁决器提供了几种优先级判别方法，利用这些方法，可以判别同时请求总线的那些主设备的优先级别，这些判别技术都假定在任一给定时刻，在发出使用总线请求的那些主设备当中，总存在一个优先级别最高的总线主设备，即它的优先级别比其它请求总线的主设备都要高。8289 的总线优先权判别方法包括并联优先权判别、串联优先权判别和循环优先权判别这几种，下面分别对之进行介绍。

在并联优先权判别方法中，系统总线上的每个裁决器都使用了一个独立的总线请求线 ($\overline{\text{BREQ}}$)，如图 12.19 所示。这些 $\overline{\text{BREQ}}$ 信号都送到一个优先权编码器中，由该编码器产生优先权最高的有效 $\overline{\text{BREQ}}$ 信号线的 2 进制地址，接着，译码器就对该 2 进制地址进行译码，以选出优先级最高且正在请求总线的裁决器所对应的 $\overline{\text{BPRN}}$ 引线，并发出相应的 $\overline{\text{BPRN}}$ 信号，当裁决器发现它获得了优先占有总线的权利时(其 $\overline{\text{BPRN}}$ 为低电平)，它就将允许其对应的总线主设备尽快获得系统总线，即一旦总线不再忙时，该主设备就将占用它。但需注意，即使某个总线主设备具有优先占有总线之特权(其优先级别最高)，它也不能立即取得总线，而必须等到当前总线传送过程的完成。一旦完成本次总线操作，总线的当前占有者就将意识到它不再具有占有总线的优先权，从而升起 $\overline{\text{BUSY}}$ 信号，释放系统总线。当 $\overline{\text{BUSY}}$ 为高电平信号时，具有占有总线优先权的那个裁决器($\overline{\text{BPRN}}$ 为低电平)就取得总线，同时，把 $\overline{\text{BUSY}}$ 置为低电平，以

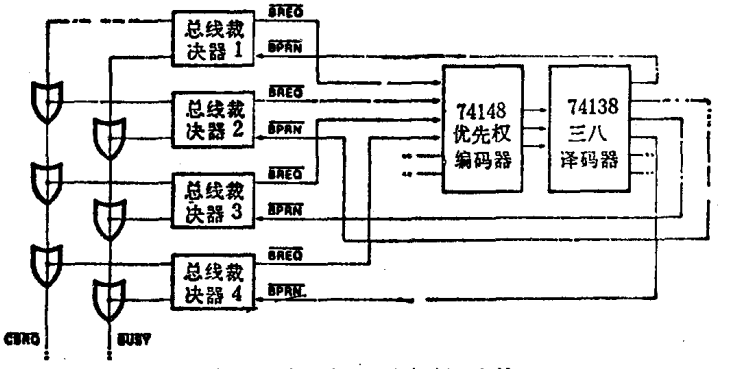
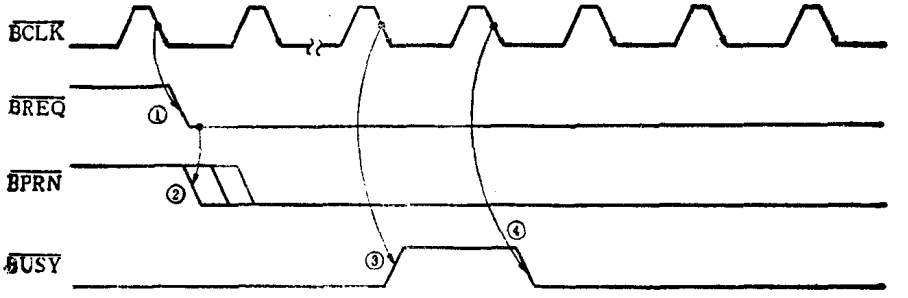


图 12.19 并联优先权判别方法



① 优先级较高的某个裁决器请求使用系统总线
② 此裁决器获得优先占有系统总线的特权
③ 优先级较低的那个总线裁决器释放总线(置不忙标志)
④ 优先级较高的那个总线裁决器获得总线并置 $\overline{\text{BUSY}}$ 为低电平

图 12.20 总线控制权的转移过程

避免其它总线裁决器占用系统总线，图 12.20 给出了获得总线过程的时序。值得注意的是，多主设备系统总线中的所有操作都是与总线时钟(BCLK)同步的。

串联优先权判别方法中不需要优先权编码器和译码器，而只是把所有的总线裁决器用菊链连到一起，即把优先级较高的总线裁决器的 $\overline{\text{BPRO}}$（总线优先权输出信号）送到下一个优先级较低的裁决器的 $\overline{\text{BPRN}}$ 端，如图 12.21 所示。

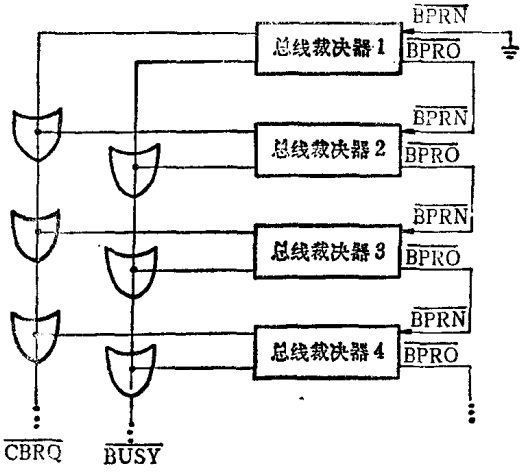


图 12.21 串联优先权判别方法

循环优先权判别方法类似于并联优先权判别方法，只是其中的优先权是动态分配的，原来的优先权编码器由一个更为复杂的电路所代替，该电路把占有总线的优先权在发出总线请求的那些裁决器之间循环移动，从而使每个总线裁决器使用系统总线的机会相同。

上面介绍的三种优先权判别方法各有其优缺点，循环优先权判别方法需要大量的外部逻辑才能实现(逻辑电路复杂)；与此相反，串联优先权判别方法不需要使用外部逻辑电路，但这种方法中所允许的总线裁决器数目受到很严格的限制，因为若总线裁决器太多，那么菊链产生的延迟就将超过系统总线时钟 $\overline{\text{BLCK}}$ 的周期长度；从一般的意义上讲，并联优先权判别方法比较好，它是其它两种方法的折衷，它允许总线上出现许多裁决器，而又不需要太多的逻辑电路来实现这种方法。

iAPX 86 系列中有两种类型的处理器，一种是 iAPX 86/10、88/10 CPU，另一种是 8089 I/O 处理器。与此相对应，8289 总线裁决器也就有两种基本操作方式，第一种是 IOB (I/O 外设总线)方式，它允许其中的处理器访问 I/O 外设总线以及系统总线；8289 的第二种工作方式是 RESB (驻留总线)方式，它允许其中的处理器通过驻留总线和系统总线进行通信。在此，所谓的 I/O 外设总线是指这样的一种总线，其上的所有设备(包括存贮器)都被当作 I/O 设备处理，都通过 I/O 命令来访问，而所有的存贮器命令则都要通过另一根总线——多个主设备的系统总线才能执行。但是，驻留总线却既能发出存贮器命令，又能发出 I/O 命令，不过，它与具有多个主设备的系统总线不同，是另外的一条总线，其区别在于驻留总线只有一个主设备，即该主设备独占此驻留总线，驻留总线对其主设备来讲总是可供使用的。

8289 的 $\overline{\text{IOB}}$ 选择信号使其以 $\overline{\text{IOB}}$ 方式工作，选择信号 RESB 又可以使相应的 8289 总线裁决器以 RESB 方式工作。当以上两个选择信号都为“假”时($\overline{\text{IOB}}$ 接高电平，RESB 接低电平)，该裁决器就仅把相应的处理器与系统总线相接，如图 12.22 所示。而当以上两个选择信号都取“真“值时($\overline{\text{IOB}}$ 接低电平，RESB 接高电平)，那么，此裁决器就把相应的处理器与多主设备的系统总线，一根驻留总线及一根 I/O 总线相连接起来。

在 $\overline{\text{IOB}}$ 方式中，处理器将通过外设总线来控制外部设备，也将通过外设总线来与外部设备通信。而当 I/O 处理器需要与系统存贮器通信时，它就要使用系统总线，图 12.23 给出了一种可能的 I/O 处理器的系统结构。

iAPX 86 及 iAPX 88 处理器能够与驻留总线及系统总线通信，在图 12.24 所示的结构当中，需要两个总线控制器，而只需要一个总线裁决器。在这种系统结构中，处理器可以访问两根总线上的存贮器及外部设备，利用存贮器映射技术来选择该访问哪根总线，裁决器上的

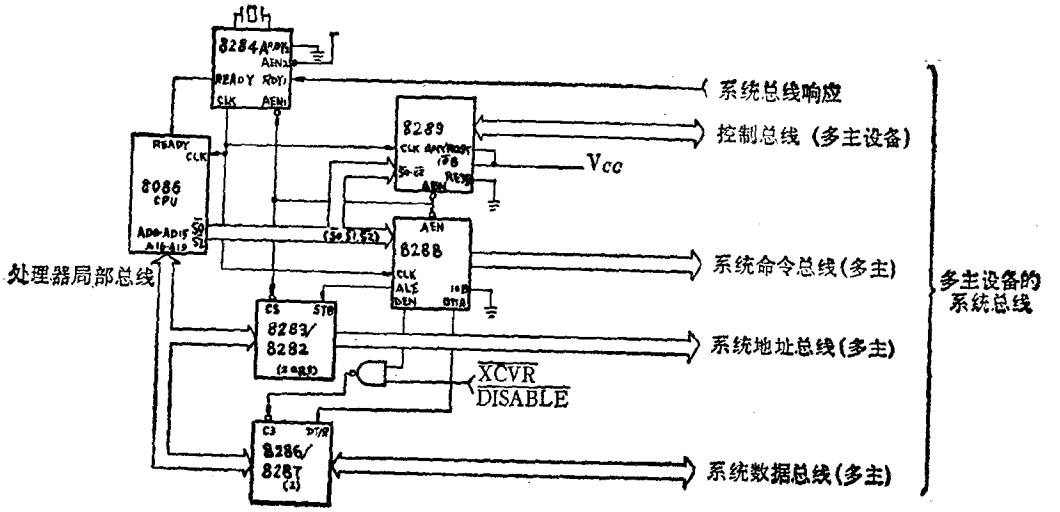


图 12.22　只包括系统总线的结构

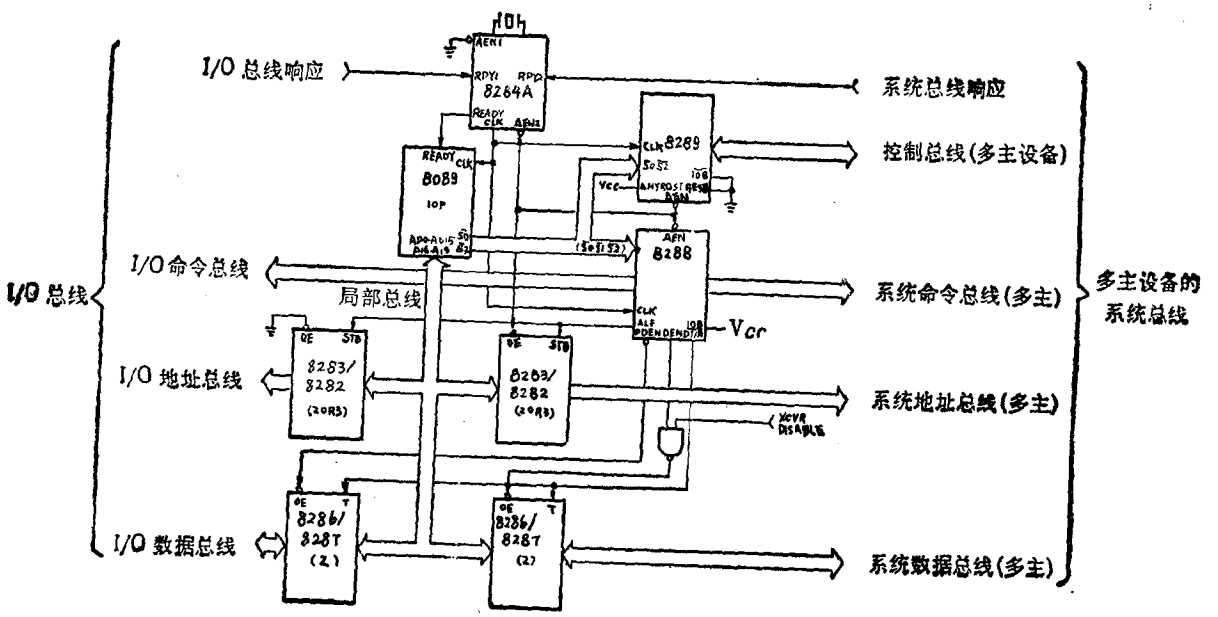


图 12.23　IOB 方式

SYSB/$\overline{\text{RESB}}$ 输入信号用于告诉该裁决器将要访问的是否为系统总线，该信号使得一个总线控制器可以发出命令，而禁止了另一个总线控制器所发出的命令。

表 12.4 总结了 8289 总线控制器所具有的几种工作方式。

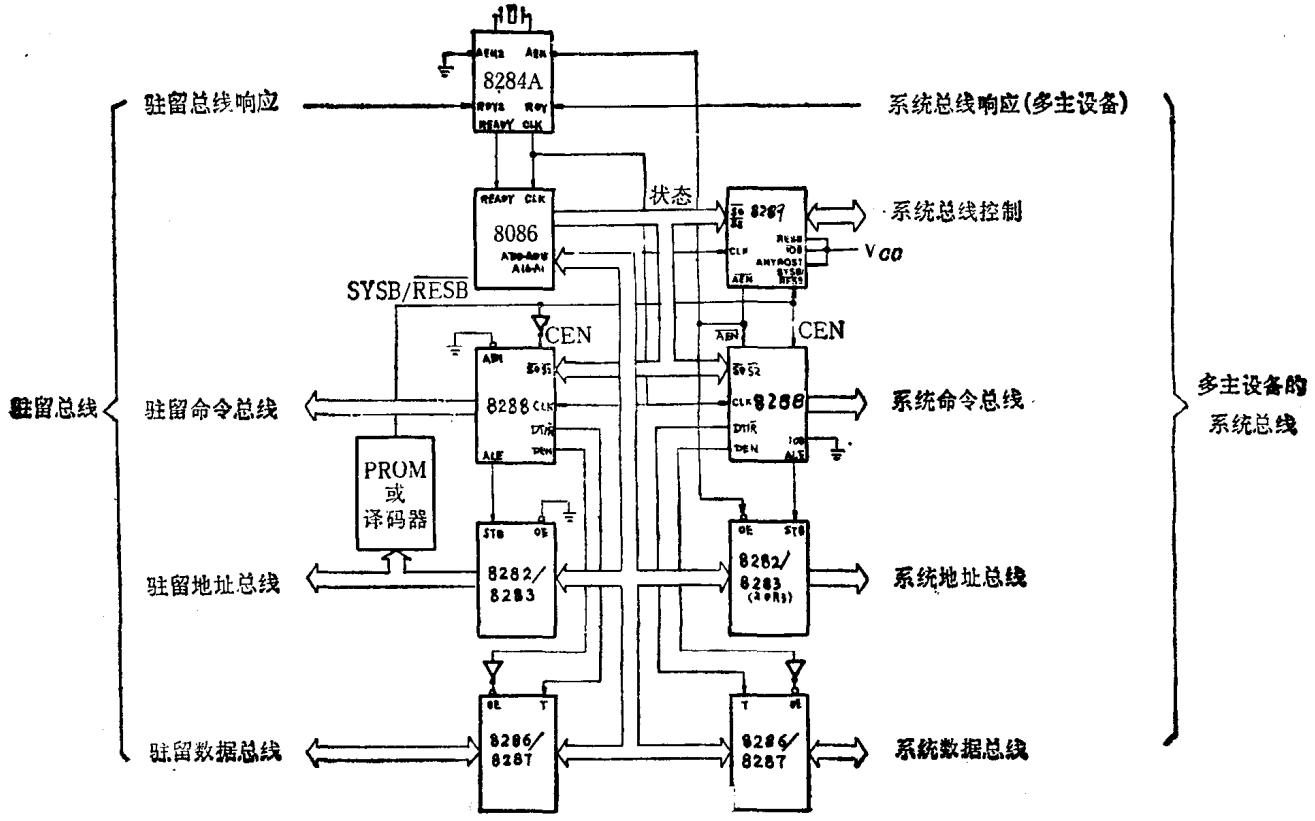


图 12.24　RESB 方式

表 12.4 8289 工作方式小结

	8086、8088、8089 的状态信号			IOB 方式	RESB 方式 $\overline{IOB}$=高电平 RESB=高电平		IOB 方式和 RESB 方式 $\overline{IOB}$=低电平 RESB=高电平		单总线方式 $\overline{IOB}$=高电平 RESB=低电平
	$\overline{S2}$	$\overline{S1}$	$\overline{S0}$	IOB=低	SYSB/RESB=高，SYSB/$\overline{RESB}$=低		SYSB/RESB=高，SYSB/$\overline{RESB}$=低		
I/O 命令	0	0	0	×		×	×	×	
	0	0	1	×		×	×	×	
	0	1	0	×		×	×	×	
HALT	0	1	1	×	×	×	×	×	×
存贮器命令	1	0	0			×		×	
	1	0	1			×		×	
	1	1	0			×		×	
空闲	1	1	1	×	×	×	×	×	×

注：×表示允许释放具有多个主设备的系统总线.

表 12.5 系统总线的请求与释放

8289 的工作方式	引脚状态	具有多个主设备的系统总线	
		请求**	释放*
单总线、多总线主方式	$\overline{IOB}$=高 RESB=低	当处理器的状态线变为有效时	HLT+TI·CBRQ+HPBRQ†
仅为 RESB 方式	$\overline{IOB}$=高 RESB=高	(SYSB/$\overline{RESB}$=高)·(有效状态信号)	(SYSB/$\overline{RESB}$=低+TI)·CBRQ+HLT+HPBRQ
仅为 IOB 方式	$\overline{IOB}$=低 RESB=低	存贮器命令	(I/O 状态+TI)·CBRQ+HLT+HPBRQ
IOB 方式·RESB方式	$\overline{IOB}$=低 RESB=高	(存贮器命令)·(SYSB/$\overline{RESB}$=高)	((I/O状态命令)+(SYSB/$\overline{RESB}$=低))·CBRQ+HPBRQ+HLT

* LOCK禁止把总线让给其它的任何裁决器，$\overline{CRQLCK}$ 禁止把总线让给任何优先级较低的裁决器.

** HALT 及无源或空闲状态除外.

† HPBRQ，优先级较高的总线请求或 $\overline{BPRN}$=1.

+代表"或"，·代表逻辑"与".

TI 表示处理器的空闲状态，即 $\overline{S2}$，$\overline{S1}$，$\overline{S0}$=111

HLT 表示处理器的暂停状态，即 $\overline{S2}$，$\overline{S1}$，$\overline{S0}$=011

§12.5 8282/8283 8位锁存器

8282 和 8283 都是双极型的 8 位锁存器，它们都具有三态的输出缓冲区。这些锁存器除了用作一般的锁存器之外，还可以用作缓冲器或多路转换器。8283 的输出数据与输入的极性相反，而 8282 中输出数据与输入数据的极性相同，这也是 8282 与 8283 的区别所在。由此可见，在微型计算机系统当中，所有主要的外设及输入/输出功能都可以使用这些锁存器。归纳起来，8282/8283 具有以下特色：

1. 可以用作 iAPX 86、88、186、188、MCS-80、MCS-85、MCS-48 诸系列中的地址锁存器；

2．具有较高的驱动能力，以驱动系统数据总线；

3．全并行的8位数据寄存器和缓冲器；

4．当选通信号有效时，其状态是透明的；

5．三态输出；

6．二十引脚的封装；

7．在进入或退出高阻抗状态时，其输出端上的噪音较低。

下面，我们准备介绍一下8283/8282锁存器的结构，图12.25给出了它们的逻辑结构，而图12.26则给出了8282/8283的引脚结构。

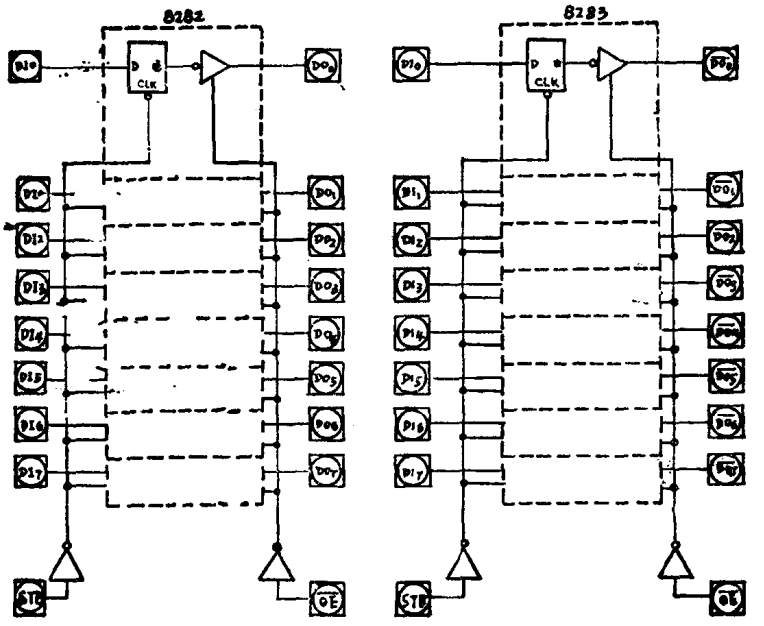


图12.25　逻辑结构框图

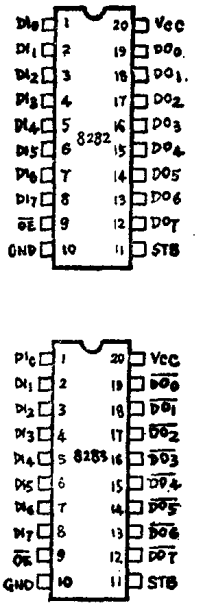


图12.26　引脚结构

各引脚的功能如下：

STB　输入信号

STB是一个起控制作用的输入信号，用于把数据输入引脚(DI_0～DI_7)上的数据选通送至数据锁存器中，故也称为“选通”信号。实际上，数据是在STB由高电平变为低电平时被锁存起来的。

$\overline{OE}$　输入信号

$\overline{OE}$也是一个起控制作用的输入信号，它所代表的意思是允许输出，即当它为低电平(有效)时，则将对应的数据锁存器中的内容送到数据输出端(DO_0～DO_7)；当$\overline{OE}$为高电平(无效)时，则将迫使其输出缓冲器处于高阻抗状态。

DI_0～DI_7　输入信号

数据输入端．出现在这些端上的数据在满足一定的建立时间之后，在STB的选通下，将被锁存器锁存起来。

DO_0～DO_7(8282)
$\overline{DO}_0$～$\overline{DO}_7$(8283)　输出信号

数据输出端．当$\overline{OE}$有效时(为低电平)，数据锁存器中的数据将被送到这些数据输出端

上，只是对 8283 来讲，输出端上的数据与输入数据极性相反，而 8282 中输出与输入的极性相同.

总起来讲，8282 和 8283 都是 8 位的锁存器，它们都具有三态的输出缓冲器. 数据在满足一定的建立时间之后，即可在选通信号 STB 由高电平变为低电平的过程中被锁存到数据锁存器中去. 在 STB 信号呈高电平的过程中，锁存器呈透明状态. 通过把 $\overline{OE}$ 信号置为有效(低电平)，锁存器中的数据就将出现在其数据输出端上；而当 $\overline{OE}$ 为高电平信号(无效)时，其输出缓冲器就为高阻抗状态.

§12.6 8286/8287 8 位总线收发器

8286 和 8287 都是具有三态输出的 8 位收发器，其间的区别在于，8287 的输出与输入数据的极性相反，而 8286 中两者的极性相同. 因此，利用它们可以满足微型计算机系统中大量的缓冲需要，即 8286/8287 可以用作系统中的缓冲驱动器.

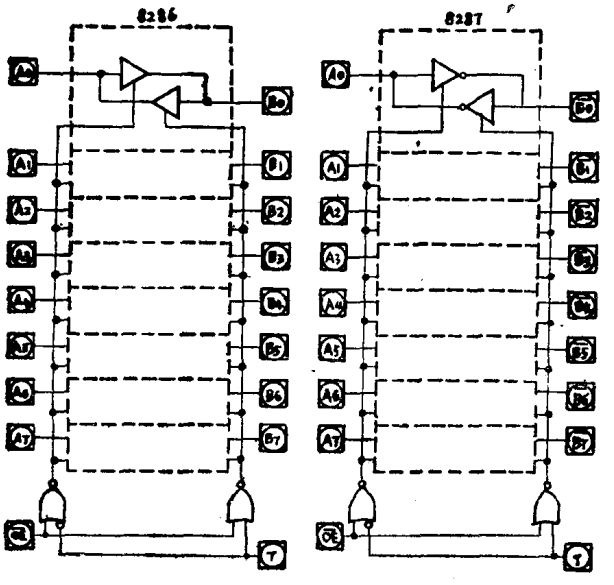


图 12.27 逻辑框图

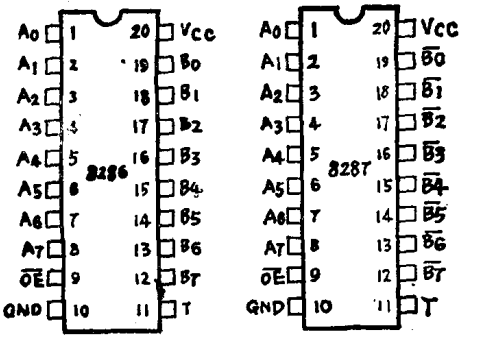


图 12.28 引脚结构

图 12.27 表示 8286/8287 总线收发器的逻辑结构框图，图 12.28 则给出了它们的引脚结构. 下面先简单地介绍一下 8286/8287 诸引脚及其相应的功能.

T 输入信号

T 是一个起控制作用的输入信号，利用该信号来控制收发器中数据的传送方向. 当 T 为高电平时，则收发器的 $B_0 \sim B_7$ 就作为输出端，而 $A_0 \sim A_7$ 为收发器的输入端；反过来，即若 T 为低电平，那么，$A_0 \sim A_7$ 就成为收发器的输出端，而 $B_0 \sim B_7$ 就用作输入端.

$\overline{OE}$ 输入信号

$\overline{OE}$ 也是一个起控制作用的输入信号，它被用来命令某个输出驱动器(由 T 确定)将其数据送到相应的总线上. 该信号低电平时代表有效.

$A_0 \sim A_7$ 输入/输出信号

局部总线数据端. 根据引脚 T 的状态，$A_0 \sim A_7$ 可以用来把数据送到处理器的局部总线上，也可以用于从局部总线接收数据.

B_0～B_7(8286)
$\overline{B}_0$～$\overline{B}_7$(8287) **输入/输出信号**

系统总线数据端。这些引脚或者用来把数据送到系统总线上，或者用来从系统总线接收数据，究竟派何用场，则要由引脚 T 的状态来确定。

由上面的引脚介绍可以看出，当 T 为有效的高电平且 $\overline{OE}$ 为有效的低电平时，引脚 A_0～A_7 上的数据将被送到引脚 B_0～B_7 上；而当 T 为低电平，$\overline{OE}$ 也为低电平时，则引脚 B_0～B_7 上的数据将被送到引脚 A_0～A_7 上。可见，8286/8287 中的数据传送是双向的，正因为如此，我们也常把 8286/8287 称做 8 位并行双向驱动器。

概括起来讲，8286/8287 具有以下特色：

1．可以用作 iAPX 86、88、186、188、MCS-80、MCS-85 及 MCS-48 系列机中的数据总线缓冲驱动器；

2．具有较高的输出驱动能力，可用来驱动系统数据总线；

3．全并行的 8 位收发器；

4．三态输出；

5．二十引脚的封装，

6．在进入或退出高阻抗状态时，输出端上无噪声。

思　考　题

12.1　8088 与 8086 相比有些什么优点？在哪些应用系统中应该使用 8088 作为 CPU？

12.2　8284A 时钟发生和驱动器具有哪些功能？它有哪两个频率源？各有什么作用？

12.3　简述 8288 的功用，$\overline{AMWC}$ 与 $\overline{AIOWC}$ 信号有何作用？

12.4　8289 是如何管理多处理器系统中的总线的？它有哪几种优先权判别方法？又有几种工作方式？

12.5　8282 与 8283 的区别与共同点是什么？

12.6　8286 与 8287 有哪些相同点，又有哪些不同点？

第十三章 MULTIBUS 系统总线

§13.1 概 述

在前面介绍 8086、8087、8088、8089、80130 的时候，我们都曾指出这些芯片是与 Intel MULTIBUS 的接口兼容的。为了使读者能够对 MULTIBUS (多总线)有个基本的了解，我们想利用一章的篇幅对 MULTIBUS 系统总线做个大致的介绍。

MULTIBUS 系统总线是 Intel 系统结构中的一种标准总线结构，其接口是一个通用的系统总线，该总线包含了多种部件之间相互作用所必需的所有信号线。在 MULTIBUS 构成的系统中，这种部件之间的相互作用是建立在主设备与从设备的概念基础之上的，借助于主、从设备之间的信息交换，从而使得速度不同的一些设备模块都能使用 MULTIBUS 系统总线接口。MULTIBUS 系统总线能够支持多个主设备，它的直接寻址能力是 16 兆字节的存贮空间。目前，MULTIBUS 已经被广泛地应用于 8/16 位微机系统之中，并逐渐成为微机总线的工业标准，它已获得了 IEEE 的确认，成为系统总线的工业标准。在实际应用中也可以见到许多以 MULTIBUS 为基础的系统，这些系统已广泛地应用于工业自动化、工业控制、办公系统、文字处理、图示设备、CAD/CAM(计算机辅助设计/计算机辅助制造)、通信系统以及分布式处理等领域当中。

英特尔公司 OEM 计算机生产线的高效率和灵活性在很大程度上得归功于 MULTIBUS 系统总线，该总线为多种系统模块之间的通信提供了方便，这些系统模块包括单板机、存贮器、I/O 扩充板、外部设备及其控制器。我们可以把 MULTIBUS 系统总线看成是 Intel 系统结构的一个物理框架，是整个系统的基础，它能够很灵活地支持 CPU、存贮器以及 I/O 设备的模块化设计，构成性能/价格比较高的微型计算机系统。系统性能的进一步提高还可以通过多机系统实现，因为 MULTIBUS 可以把多个单板机及其扩充模块连接起来，故利用 MULTIBUS 系统总线，就能够很容易地组成多机系统，而在这种多机系统环境下，我们可以把任务分成许多较小的子任务，再由各个处理机分别执行各自的子任务，从而获得系统中任务的并行执行，这样，既提高了完成任务的速度，又提高了系统的性能。

MULTIBUS 系统总线的特色可以归纳为下面几条：

(1) IEEE796 系统总线的工业标准；

(2) 支持多处理机系统，它是一个具有多个主设备的总线；

(3) 8 位和 16 位设备都可以共享 MULTIBUS 的系统资源；

(4) 它可以作为整个 Intel 系统结构的基础；

(5) 16 兆字节的寻址能力；

(6) 总线带宽最高可达每秒 10 兆字节；

(7) 单板机、存贮器、数字 I/O 设备、模拟 I/O 设备、外设控制器等都可以挂到 MULTIBUS 系统总线上。

§13.2 MULTIBUS 系统总线的结构

MULTIBUS 接口是一个采用异步通信方式的、具有多重处理特色的系统总线，设计这样一个系统总线的目的就是为了在单板机、存贮器和 I/O 扩充板之间实现 8 位和 16 位的信息传输功能。它由 24 根地址线、16 根数据线、12 根控制线、9 根中断线以及 6 根总线交换线所组成。用来与 MULTIBUS 系统总线相接的印刷电路板(6.75″×12.00″)有两个插头，把它插入总线底板之后，就相当于实现了与 MULTIBUS 的接口。两个插头是 86 根引线的 P_1(主插头)和 60 根引线的 P_2(辅助插头)，主插头 P_1 中包含了除 4 根地址线之外的所有的 MULTIBUS 信号线，而辅插头 P_2 主要是用于进行系统的扩充，P_2 中含有 4 根扩充的地址线。

以多总线 MULTIBUS 为基础的系统中的各模块之间具有主/从关系，归纳起来讲，MULTIBUS 系统总线能够对三种类型的设备提供支持，这三类设备是：(1)主设备；(2)从设备；(3)智能从设备。

总线主设备是指这样一些模块，它们具有控制总线的能力，可以驱动命令线和地址线，系统中的单板机就属于这种模块。一般来讲，系统中的主设备不止一个，即有多个模块具有控制总线的能力，MULTIBUS 接口是通过一个总线交换逻辑来满足这种需要的。当系统中有几个主设备同时请求控制总线时，就要进行总线仲裁，通常是由某个总线主设备提供总线时钟信号，总线时钟与处理器时钟没有关系，它只是提供一个定时基准，以解决总线主设备之间的总线竞争问题。总线的信息传输速度仅仅依赖于传送设备和接收设备。一旦总线控制权由某个主设备所获得，该主设备就可以发出命令信号、地址信号以及存贮器或 I/O 地址，对数据传输进行控制，在此期间，其它总线主设备就不可能获得总线控制权。

总线从设备是这样的一些模块，它们不能控制 MULTIBUS 系统总线接口，而只能对多总线上的地址进行译码，或者根据总线主设备所发来的命令信号进行一些操作。系统中的存贮器以及 I/O 设备就是处于这种地位的总线从设备。

智能从设备与上面所讲的从设备具有相同的总线接口特性，但它本身带有微处理器，即它自己能够执行一些操作，片上的这种处理器、存贮器和 I/O 组合使得智能从设备能在不使用 MULTIBUS 系统总线的情况下完成一些任务。

按照各信号线所起的作用，我们可以把 MULTIBUS 系统总线的所有信号线分为五类，即 1) 控制线；2) 地址和禁止线；3) 数据线；4) 中断线；5) 总线交换线。图 13.1 给出了 MULTIBUS 的这些信号线。

下面，我们就对 MULTIBUS 的诸信号线分别进行介绍。值得注意的是，MULTIBUS 中的绝大多数信号都是低电平有效的，即地址或数据总线上的低电平代表逻辑 "1"，而控制信号线上的低电平则表示该控制信号有效。在后面的讨论中，我们准备通过在各信号助记符的后面加一斜杠(/)，以表示该助记符所代表的信号是低电平有效的。

INIT/：初始化信号。该信号将整个系统复位到一个已知的内部状态，它可由某个总线主驱动，也可以由某个外部信号源所设置。

ADR0/-ADR17/：二十四根地址线。用于传送所要访问的存贮器单元或 I/O 转接口的地址。这些地址线是通过 ADR0 到 ADR9/、ADRA/到 ADRF/以及 ADR10/到 ADR17/来标识的，即采用的是 16 进制的编号，其中，ADR17/是地址线的最高有效位。这二十四根地址

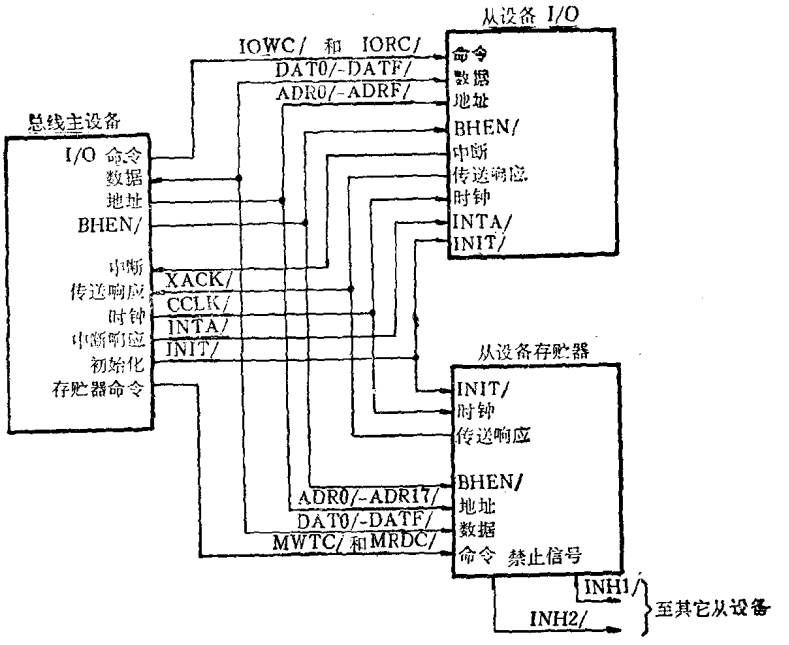


图 13.1 MULTIBUS 接口信号线

线使得系统可以访问 16 兆字节的存贮器空间，当所访问的是 I/O 设备时，则仅使用其中的十六根地址线，即系统最多可以访问 64 K 的 I/O 空间。但实际上，对于 8 位的主设备来讲，它只用了十六根地址线 (ADR0/～ADRF/) 来访问存贮器，只用了八根地址线 (ADR0/～ADR7/) 来寻址 I/O 转接口，即 8 位主设备仅可访问 64 K 字节的存贮空间和 256 个 I/O 设备。而对于 16 位的主设备(如 8086 CPU)来讲，它可以把二十根地址线(ADR0/～ADR13/)用于寻址存贮器，把十二根地址线 (ADR0/～ADRB/) 用于选择 I/O 转接口，故 16 位主设备可以访问 1 兆字节的存贮空间和 4096 个 I/O 设备。

BHEN/：这是一个字节控制信号。利用它来规定数据在 MULTIBUS 数据总线的高字节部分(DAT8/～DATF/)进行传送。该信号只有在 16 位存贮器和 I/O 模块的系统中用到。

INH1/：禁止 RAM 信号。它使得 RAM 存贮器设备不响应地址总线上的存贮器地址。当 ROM 和 RAM 存贮器地址相同时，INH1/信号实际上就是表明该地址选中的是 ROM 上的相应单元，而不是 RAM 中的单元。本信号还可以用于使存贮器映象的 I/O 设备取代 RAM 存贮器。

INH2/：禁止 ROM 信号。它使得 ROM 存贮器不响应系统地址总线上的存贮器地址。

DAT0/～DATF/：十六根双向数据线。它们用于在主设备与存贮器单元或 I/O 转接口之间传送信息。DATF/是数据的最高有效位。8 位系统只用到 DAT0/～DAT7/这八根数据线，其中 DAT7/是数据的最高有效位；在 16 位系统中，数据传送可用八根数据线，也可以用十六根数据线。

BCLK/：总线时钟。BCLK/的下降沿可用来与总线优先权判别电路同步，该时钟信号并不与 CPU 的时钟同步，其最短周期是 100 ns。为了便于调试，可以使 BCLK/放慢、停止或以单步方式发出信号。

CCLK/：固定时钟。该总线信号提供了一个固定频率的时钟信号，其最短周期亦为 100 ns。

BPRN/：总线优先权输入信号。它告诉某个特定的主设备，系统中不存在优先级更高的主设备在请求使用总线，该信号是与BCLK/同步的，它在总线的底板上不占位置。

BPRO/：总线优先权输出信号。它用于串联(菊链)总线优先权选择模式中，各总线主设备按照其优先级的高低串行地连接起来，即把本主设备的 BPRO/信号接到下一个总线优先权较低的主设备的BPRN/输入端。BPRO/信号也是与 BCLK/同步的。

BUSY/：总线忙信号。它是由当前控制总线的主设备所驱动的，该信号有效时表示正在使用总线，从而，BUSY/信号能够阻止所有其它的主设备获得总线控制权，它与 BCLK/信号同步。

BREQ/：总线请求信号。该信号用于并联的总线优先权网络中，它表示某个主设备需要利用总线进行数据传输。BREQ/与BCLK/信号同步。

CBRQ/：公共总线请求信号。它由所有可能的总线主设备所驱动，用来通知当前的总线主设备另有一个主设备希望使用总线。若 CBRQ/为高电平，则表示没有别的主设备在请求使用总线，所以，当前的总线主设备将保留总线控制权。

MRDC/：读存贮器命令。该信号表明存贮器单元的地址已经放在系统的地址线上，且要求把该单元中的内容 (8 位或 16 位) 读出，并放在系统的数据线上。这里需要解释一下，MULTIBUS 协议规定：总线主设备至少必须在发出读/写命令的 50 ns 之前送出地址信号(对于写命令，还需同时提前送出欲写的数据)，从而保证在执行读/写操作时，地址(对写操作还包括数据)是有效的，即地址(和数据)处于稳定状态。基于同样的理由，协议规定至少在发出读/写命令 50 ns 之后，才可以改变地址线上的信号。信号 MRDC/并不需要与 BCLK/保持同步。

MWTC/：写存贮器命令。它表示存贮器单元的地址以及将写入这个单元的数据都已经分别出现在系统的地址线和数据线上，该命令要求把数据线上的数据写到所寻址的存贮器单元中去。MWTC/不需要与BCLK/保持同步。

IORC/：读 I/O 命令。它表明输入转接口的地址已经出现在系统的地址总线上，而该输入转接口的数据将被读到系统的数据总线上。IORC/信号也不需要与 BCLK/同步。

IOWC/：写 I/O 命令。它表明输出转接口的地址已被放到系统的地址总线上，而系统的数据总线上的数据 (8 位或 16 位)将被输出到所寻址的输出转接口。该信号也无需与 BCLK/保持同步。

XACK/：传送响应信号。这是从设备送给总线主设备的一个响应信号，它表示读/写操作已告完成，也就是说，数据已经放到了系统的数据总线上，或已经从数据总线上接收到数据。XACK/信号也不与 BCLK/同步。

INT 0/～INT 7/：八根中断请求线。它们分别代表八个级别的中断请求，其中，INT 0/的优先级最高，而 INT 7/的优先级最低。这些中断请求信号供并联的中断分解网络所使用。

INTA/：中断响应。中断响应信号是由总线主设备发出的，它要求中断控制器 (8259 或 8259 A)把中断信息送到总线上，即把中断向量的地址送到数据总线上。该信号是在主设备接收到中断请求之后发出的。

LOCK/：封锁信号。该信号是由掌握总线控制权的主设备用来阻止其它总线主设备占用总线的。

备用引线：插头 P_1 和 P_2 上都有一些没有使用到的引线端，我们可以把这些引线看作是留待

今后使用的。

电源：见表 13.1。

表 13.1 MULTIBUS 系统总线的引线/信号分布(P_1)

	引脚号	助记符	解释	引脚号	助记符	解释
电源	1	GND	接地	2	GND	
	3	+5 V		4	+5 V	
	5	+5 V		6	+5 V	
	7	+12 V		8	+12 V	
	9	−5 V		10	−5 V	
	11	GND		12	GND	
总线控制	13	BCLK/	总线时钟	14	INIT/	初始化
	15	BPRN/	总线优先权输入	16	BPRO/	总线优先权输出
	17	BUSY/	总线忙	18	BREQ/	总线请求
	19	MRDC/	存贮器读命令	20	MWTC/	存贮器写命令
	21	IORC/	I/O 读命令	22	IOWC/	I/O 写命令
	23	XACK/	传送响应	24	INH1/	禁止 RAM
总线控制与地址	25		保留	26	INH2/	禁止 PROM 或 ROM
	27	BHEN/	允许高字节	28	AD10/	地址总线
	29	CBRQ/	公共总线请求	30	AD11/	
	31	CCLK/	常时钟	32	AD12/	
	33	INTA/	中断响应	34	AD13/	
中断	35	INT6/	并联的中断请求	36	INT7/	并联的中断请求
	37	INT4/		38	INT5/	
	39	INT2/		40	INT3/	
	41	INT0/		42	INT1/	
地址	43	ADRE/	地址总线	44	ADRF/	地址总线
	45	ADRC/		46	ADRD/	
	47	ADRA/		48	ADRB/	
	49	ADR8/		50	ADR9/	
	51	ADR6/		52	ADR7/	
	53	ADR4/		54	ADR5/	
	55	ADR2/		56	ADR3/	
	57	ADR0/		58	ADR1/	
数据	59	DATE/	数据总线	60	DATF/	数据总线
	61	DATC/		62	DATD/	
	63	DATA/		64	DATB/	
	65	DAT8/		66	DAT9/	
	67	DAT6/		68	DAT7/	
	69	DAT4/		70	DAT5/	
	71	DAT2/		72	DAT3/	
	73	DAT0/		74	DAT1/	
电源	75	GND	保留	76	GND	保留
	77			78		
	79	−12 V		80	−12 V	
	81	+5 V		82	+5 V	
	83	+5 V		84	+5 V	
	85	GND		86	GND	

表 13.2 MULTIBUS 的引线/信号分布(P_2)

	引脚号	助记符	意思	引脚号	助记符	意思
	1			2		
	3			4		
	5			6		
	7			8		
	9			10		
	11			12		
	13			14		
	15			16		
	17		保留	18		保留
	19			20		
	21			22		
	23			24		
	25			26		
	27			28		
	29			30		
	31			32		
	33			34		
	35			36		
	37			38		
	39			40		
	41			42		
	43			44		
	45			46		
	47		保留	48		保留
	49			50		
	51			52		
	53			54		
地址	55	ADR16/	地址总线	56	ADR17/	地址总线
	57	ADR14/		58	ADR15/	
	59		保留	60		保留

§13.3 MULTIBUS 系统总线的操作原理

§13.3.1 数据传送操作

MULTIBUS 系统总线的数据传送操作是以异步方式进行的，它遵守主——从信息交换协议，即以一般的应答方式实现总线上的数据传送过程。首先，总线主设备必须把存贮器单元或 I/O 转接口的地址放到地址总线上，若为写操作，则还需要同时把数据放到数据总线上；然后，总线主设备发出数据传送命令 (I/O 读/写，或存贮器读/写命令)，启动相应的总线从设备，真正开始数据传送操作；若为写命令，则从设备就将接收数据总线上的数据，若为读命令；那么，从设备就将把数据送到数据总线上；然后，从设备就给总线主设备发出传送响应信号，这样，主设备就将完成本次读/写周期，从命令线上移去命令，再从总线上移去地址和数据信息。

图 13.2 给出了读数据操作的时序图。地址至少必须在 IORC/或 MRDC/命令发出之前

50 ns (t_{AS}) 发出，以保证在执行读操作时，地址信号是有效的，这一段时间由总线接口用来对地址进行译码，并由此给出所需要的设备选择信号，建立相应的数据通路。命令信号起码得保持 100 ns (t_{CMD} 的最小值)，在命令被撤消之后，地址还必须至少保持 50 ns(t_{AH})。数据不应在命令之前送出，但必须保持到命令撤消之后才能移去。响应信号 XACK/表明规定的读/写操作已经完成，显然，它必须在读命令及有效数据发出之后才能出现，且需保持有效到命令撤消之后。

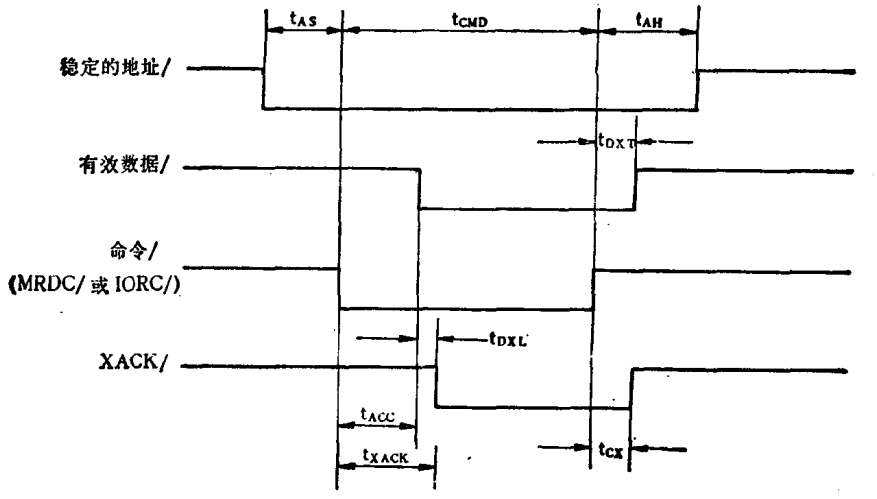


图 13.2　读数据操作的时序

写操作的时序如图 13.3 所示。在进行写操作时，地址和数据都必须稳定地出现在各自的总线上，因此，对数据的建立时间 (t_{DS}) 与对地址的建立时间 (t_{AS}) 的要求是相同的。总线协议要求在写命令前后，数据都要保持稳定，这就使得总线接口电路能在命令的前沿或后沿锁存数据，从而给系统的设计提供了更大的灵活性。

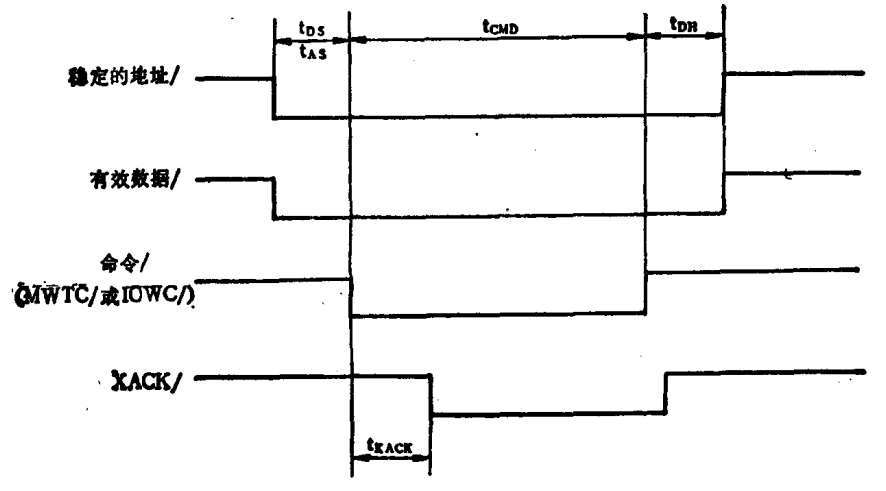


图 13.3　写数据操作的时序图

在以 MULTIBUS 系统总线为基础所构成的系统中，16 位的总线主设备可以用 8 位或 16 位数据通路，至于采用 8 位通路还是 16 位通路则要取决于所需执行的是字节操作还是字(两个字节)操作。对于 8 位的总线主设备而言它仅可在 MULTIBUS 的数据线 DAT 0/DAT 7/上进行字节传送；但对 16 位的主设备来讲，它可以进行字传送操作，也可以进行字节传送操作，

这是由 I/O 或存贮器地址的奇偶性决定的。

为了与原有的 8 位主设备和从设备保持兼容，在所有新一代的 16 位主设备及从设备中都带有一个字节交换缓冲器。图 13.4 给出了一种用于 16 位主设备和从设备系统中的 8/16 位数据驱动逻辑，在该 8/16 位系统中，存在三套缓冲器，即存取 DAT0/～DAT7/的低位字节缓冲器、存取 DAT8/～DATF/的高字节缓冲器以及用于字节交换的字节交换缓冲器，字节交换缓冲器存取多总线数据线 DAT0/～DAT7/上的数据，并实现该数据与数据线 DAT8/～DATF/来回传送。

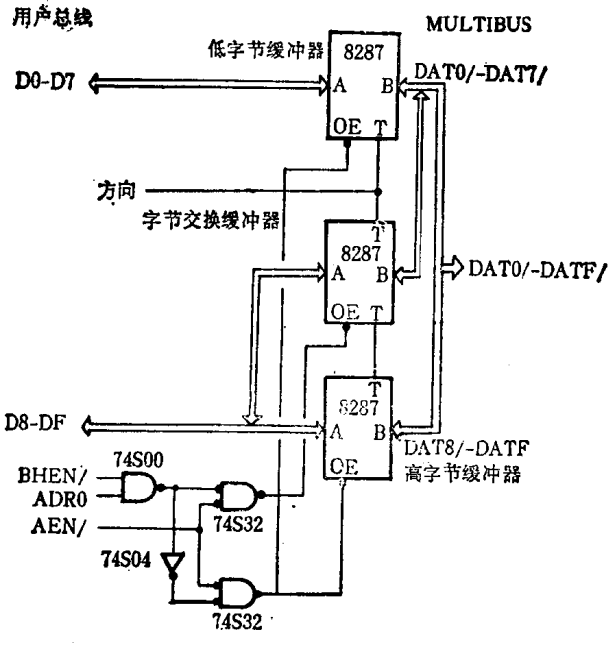


图 13.4 8/16 位数据驱动器

图 13.5 对三种类型的多总线传送中所使用的 8 位或 16 位通路情况进行了小结，由图中可见，MULTIBUS 系统总线中的数据传送受到两个信号(BHEN/和 ADR0/)的控制。

对于第一种类型的数据传送，BHEN/和 ADR0/均为高电平，这时它传送的是一个位于偶数地址的字节数据，其传送操作是在数据线 DAT0/～DAT7/上完成的；对第二种类型的数

16 位 的 设 备	MULTIBUS	BHEN/	ADR0/	MULTIBUS 数据传送通路	传送的设备字节
低、偶字节	DAT0/-DAT7/	高 电 平	高 电 平	8 位：DAT0/～DAT7/	偶 字 节
高、奇字节	DAT8/-DATF/				
低、偶字节	DAT0/-DAT7/	高 电 平	低 电 平	8 位：DAT0/～DAT7/	奇 字 节
高、奇字节	DAT8/-DATF/				
低、偶字节	DAT0/-DAT7/	低 电 平	高 电 平	16 位：DAT0/～DATF/	偶字节和奇字节
高、奇字节	DAT8/-DATF/				

图 13.5 8/16 位设备的传送操作

据传送而言，BHEN/为高电平(无效)，ADR0/为低电平(代表"1")，这时它传送的是位于奇地址的高字节数据，该字节数据通过字节交换缓冲器被送到数据线 DAT0/～DAT7/上，这就为8位系统与16位系统之间的兼容提供了可能性；第三种类型的传送是一种字传送，即每次传送16位，当采用这种字传送方式时，BHEN/为低电平(有效)，而 ADR0/为高电平(代表"0"，即为偶地址)，数据线 DAT0/～DAT7/上传送的是低字节(位于偶地址)，数据线 DAT8/～DATF/ 上传送高字节(位于奇数地址)；至于 BHEN/和 ADR0/都为低电平的情况图11.5中没有列出，我们可以把它用于在 DAT8/～DATF/上传送数据的高字节部分，显然，这时不需要使用字节交换缓冲器，由于这种情况下不能与8位模块进行通信，故一般不提倡采用这种传送方式。

§13.3.2 中断操作

MULTIBUS 系统总线中的主设备利用中断信号线 INT0/～INT7/接收中断，其中断源可以是总线从设备、其它的总线主设备或象电源失效之类的外部逻辑电路，对于总线主设备来讲，它也有可能包含了某些外部中断源，只是这些中断源不需要使用总线的中断信号线来中断该主设备。MULTIBUS 系统总线的接口支持两类中断处理方式，即支持非总线向量中断和总线向量中断。当总线主设备对非总线向量中断进行处理时，不需要多总线接口传送该中断向量的地址；但向量中断却与此不同，它需要把中断向量的地址沿数据总线由从设备送到主设备，为了保持同步，该过程中用到了中断响应命令信号。中断向量地址是由中断控制器产生的。当中断请求发生时，总线主设备上的中断控制逻辑就将中断处理机，并发出中断响应命令，从而让总线上的中断逻辑去判别中断的优先级别，这时将封锁系统总线。在总线主设备选出优先级最高的中断请求之后，总线从设备就将把其中断向量的地址送到数据总线上，这个地址是指向该中断对应的处理程序的。

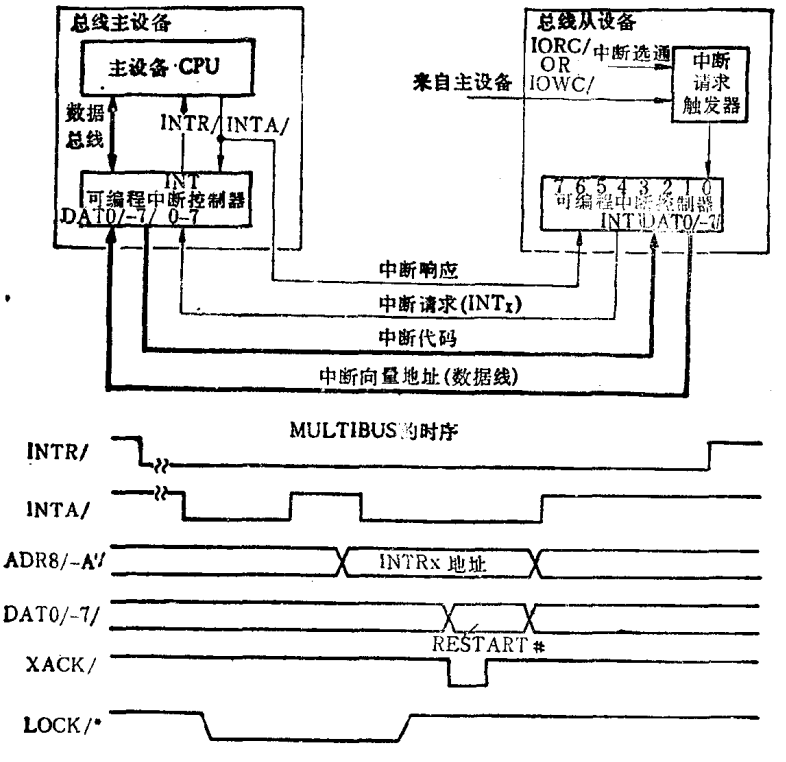
非总线向量中断是这样的一些中断，它们的中断向量地址是由总线主设备产生的，这些中断向量的地址不需要通过 MULTIBUS 的地址线。实际上，中断向量地址是由主设备上的中断控制器所产生，并通过局部总线传送到处理器。中断源可以是主设备，也可以在其它总线模块上，在后一种情况下，这些总线模块(中断源)将通过 MULTIBUS 的中断请求线 INT0/～INT7/向总线主设备发出相应的中断请求信号，当该总线主设备接到中断请求信号之后，它将执行自己的中断操作，并对此中断进行处理。实际上我们这里所讲的总线主设备主要是指主 CPU 与中断控制器的一个组合，从设备的中断请求信号通过 INT0/～INT7/送给可编程的中断控制器(如8259)，至于中断向量的地址则是由中断控制器形成并送给 CPU 的。

总线向量中断需要通过 MULTIBUS 的数据线将中断向量的地址由从设备送到总线主设备，同时利用 INTA/命令信号实现同步。当 INT0/～INT7/上有一个中断请求信号出现时，总线主上的中断控制逻辑就将中断其处理器，该总线主上的处理器接着发出 INTA/命令，该命令将使多总线上的从设备保持其中断逻辑的状态，以便后面进行优先级的判别，该总线主设备为了继续占用总线，还要锁住 MULTIBUS 的控制线。在第一个 INTA/命令发出之后，总线主设备的中断控制逻辑把一个中断码放到 MULTIBUS 的地址线 ADR8/～ADRA/上，该中断码是当前优先级最高的中断请求线的地址。在总线向量中断的处理过程中，有可能出现两种不同的中断处理过程，之所以会出现这种现象，是因为 MULTIBUS 中的主设备既可以是

8086 也可以是 8080 A 或 8085 的。对于前者，即总线主设备为 8086 时，该主设备将再发出一个 INTA/信号，这就使得总线从设备的中断控制逻辑把一个 8 位的中断向量指针送到 MULTIBUS 的数据线上，而总线主设备就可以利用这个向量指针来确定其中断服务子程序的地址；对于后一种情况，即总线主设备为 8080 或 8085 时，该总线主设备将再发出两个 INTA/信号，这两个 INTA/命令就使得总线从设备可以把两字节的中断向量地址送到 MULTIBUS 系统总线的数据线上，即每发一个 INTA/命令，则传送一个字节。而总线主设备就可以利用该中断向量地址找到对应的中断服务子程序，对中断进行处理。

但是应该注意，在一特定的系统当中，只可以使用一种总线向量中断操作方式，这是由 MULTIBUS 协议所规定的。具体讲就是，若总线从设备上的中断控制器是 8259，则从设备板仅可以进行 3 个 INTA/的操作方式(上面介绍的第二种情况。假如从设备上的中断控制器是 8259 A，那么它就能够采用两个 INTA/或三个 INTA/的操作，但是，如果系统采用的是总线向量中断的话，那么，该系统中的所有从设备就都必须采用相同的中断操作方式，即要么全用二 INTA/，要么全用三 INTA/。然而，在同一系统中既可以用总线向量中断，又可以用非总线向量中断。

图 13.6 给出了总线向量中断逻辑结构以及二 INTA/信号的定时序列。



*LOCK/实际上并不是 MULTIBUS 的信号

图 13.6 总线向量中断的逻辑结构(采用二 INTA/方式)

§13.3.3 总线交换技术

前已述及，MULTIBUS 系统总线的主设备可以不止一个，也就是说，在同一系统中，可以

存在多个总线主设备。当其中的某个总线主设备需要进行数据传送时，它就将试图取得总线控制权。这些总线主设备是通过总线交换技术来请求并获得总线的。

MULTIBUS 系统总线的接口提供了两种总线交换的优先权技术，即串联优先权技术和并联优先权技术。串联优先权技术相当于一般的菊链式结构，即系统总线请求的优先级别是由对应的主设备的位置所确定的。如图 13.7 所示，在这种菊链式结构中，优先级别最高的主设备的优先级输入端(BPRN/)是与地相接的，而它的优先级输出端(BPRO/)则与下一个优先级较低(次最高)的主设备的优先级输入端相连，如此下去，将所有的主设备端按照其优先级的高低顺序连成一条链。若链中的某个总线主设备发出总线请求信号，那么，它就将把其 BPRO/信号置为高电平，该信号又传递到下一个优先级较低的主设备的 BPRN/端，而对于任何主设备来讲，它一旦发现其 BPRN/信号为高，就把它的 BPRO/也置为高，这样，一直传递到优先级最低的那个主设备。通过这一过程，也就确定了当前请求总线的主设备中的优先级最高者。但很显然，当采用串联优先权技术时，总线主设备的个数受到限制，这是由于整个菊链延迟时间太长的缘故。例如，当 BCLK/的周期为 100 ns 时，至多只能使用三个总线主设备。假如需要使用更多的主设备，那么，就必须延长 BCLK/的周期或采用并联优先级技术。

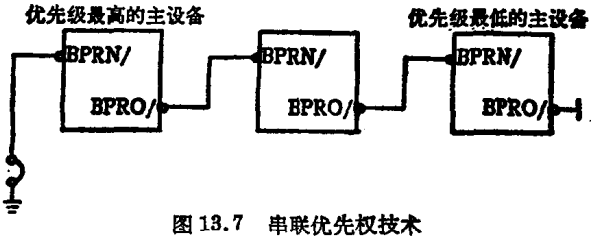


图 13.7　串联优先权技术

当采用并联优先权技术时，其优先级别是通过一个优先权判别电路分解的。电路中利用一个优先权编码芯片(74148)对优先级最高的 BREQ/输入信号进行编码，然后，优先权译码芯片(74S138)再对该编码值进行译码，并触发相应的 BPRN/信号线。在这种并联的优先权方案中没有使用 BPRO/信号。但也应注意，由于物理总线的长度受到限制，故当用并行优先权技术时，系统中最多也只可以使用 16 个总线主设备。图 13.8 给出了一种较具代表性的并行优先权分解网络。

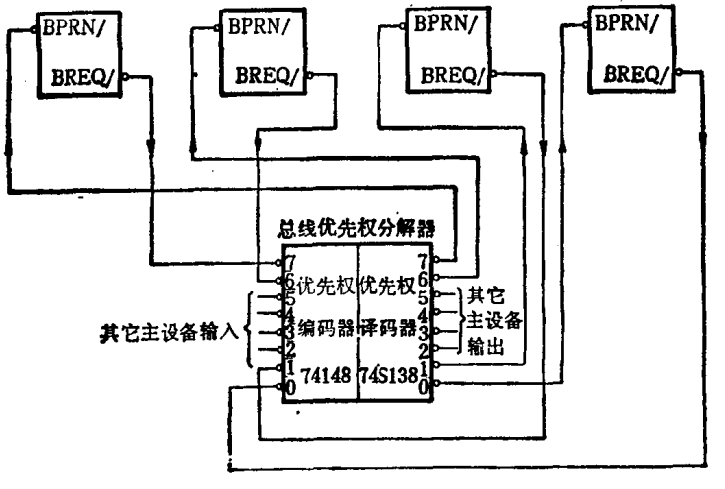


图 13.8　并联优先权技术

图13.9 表示出了一种用于进行 MULTIBUS 总线交换操作的时序，本例采用的是并联优先权判别方案，然而，对于串联优先权分解方案来讲，其总线交换的时序过程也基本上与此相同。

在图 13.9 给出的例子中，我们假定主设备 A 的优先级别比主设备 B 低，因为只有这样，当主设备 A 占用总线控制权期间，主设备 B 请求使用总线时，才有可能发出总线交换操作。

在主设备 A 控制总线期间，如果总线主设备 B 需要使用总线，以访问 I/O 或存贮器模块，

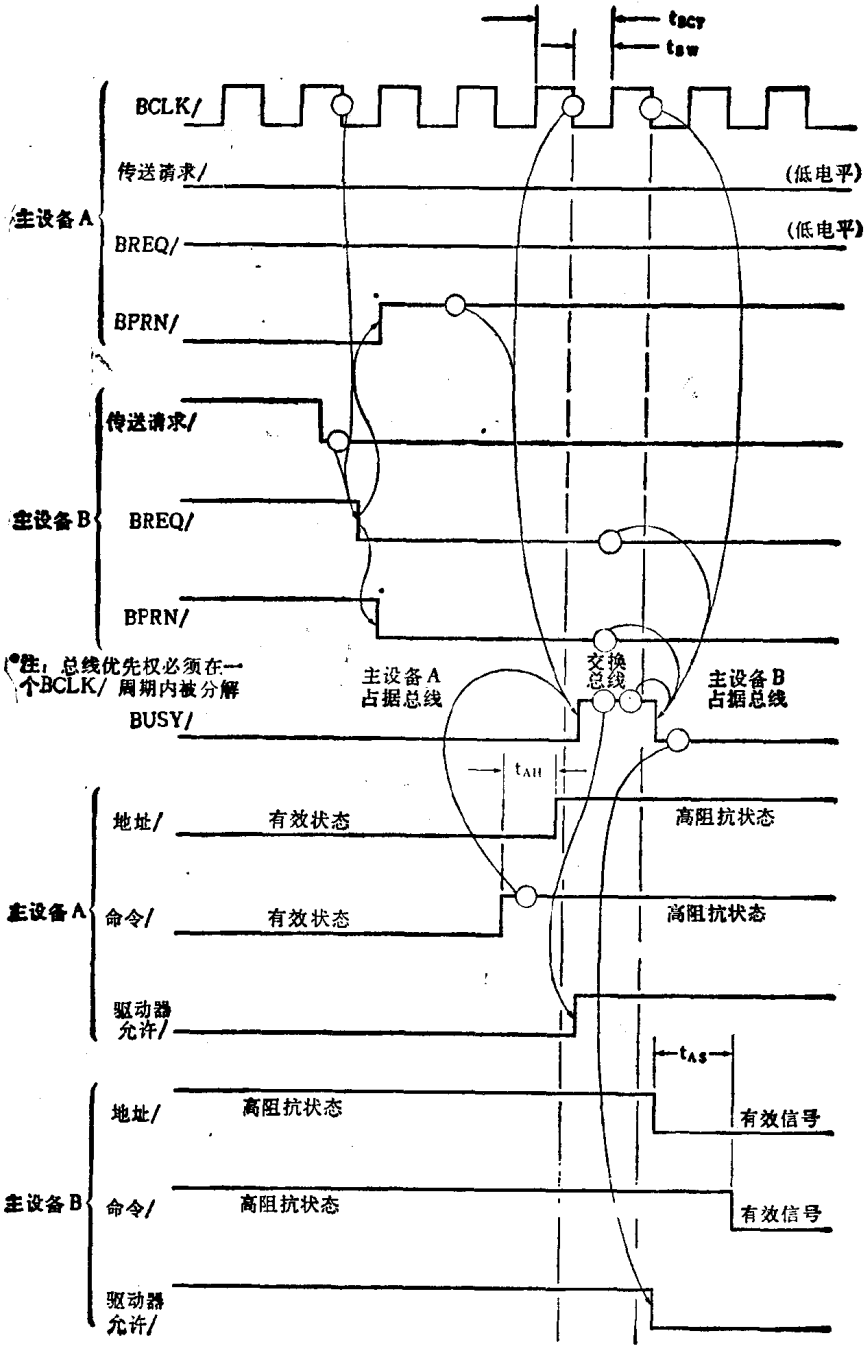


图 13.9　总线交换的操作过程

那么我们就称总线交换过程开始了。该传输请求信号与 BCLK/的下降沿一起同步地产生一个总线请求信号 (BREQ/)，这时，总线优先权判别电路就把主设备 A 的 BPRN/信号由低电平变为高电平（有效变为无效），而把主设备 B 的 BPRN/信号由高电平变为低电平（无效变为有效）。在主设备 A 完成了当前命令的操作之后，它就将在下一个 BCLK/的下降沿把 BUSY/信号置为无效（高电平），这就允许开始实际的总线交换过程了，因为此时主设备 A 已经放弃了总线的控制权，而总线主设备 B 的请求已经得到响应（BPRN/信号证实了这一点）。至此，总线主设备 A 的驱动器被禁止，而主设备 B 将在下一个 BCLK/的下降沿取得总线控制权，即把 BUSY/信号置为有效，且允许主设备 B 的驱动器进行工作。

当然，主设备 A 也有可能需要保持总线的控制权，而不让主设备 B 取得总线，为了达到这一目的，总线主设备 A 只要发出一个总线封锁信号就可以了，该信号将使得 BUSY/保持有效，即不让 BUSY/变为高电平，从而也就使得总线控制权总是掌握在主设备 A 手中。由此可见，借助于总线封锁信号，就可以为某个总线主设备连续地占有一些总线周期提供保障，这对于实现那些需要连续地访问总线的软件或硬件功能是颇有益处的。

在仅仅包括一个总线主设备的系统中，为了能够访问某些从设备，就有必要把该主设备的 BPRN/引脚接地。在那些使用了能够进行总线向量中断操作的 CPU 的单板系统当中，BPRN/引脚也必须接地。在仅包含一个主设备的系统中，如果保持总线封锁信号为有效，那么就可以获得比较高的传输效率，因为总线封锁信号将使该主设备在所有的时间内都保持着总线控制权，因而，不再需要为获得总线而花费任何时间。

对于当时正控制着总线的主设备来讲，它可以利用 CBRQ/信号线来确定系统中是否存在着其它的主设备正在请求使用总线。当控制着总线的主设备发现 CBRQ/信号处于无效状态时，它就将继续保持总线控制权，因此，如果该总线主设备需要再次访问总线，那么，它就不再需要为了重新获得总线而另花时间；反之，即若当前的总线主发现 CBRQ/信号为有效的话，那么，在当前的总线访问操作结束之后，它就将释放总线的控制权，而与需要使用总线的主设备一道争用总线，这时，主设备相应的优先权确定了总线控制权的所有者。

思 考 题

13.1 MULTIBUS 由哪些信号线组成？

13.2 规定系统总线的标准有些什么好处？

13.3 简述多总线上的数据传送过程？地址为何要在读/写操作之前发出？

13.4 MULTIBUS 系统总线上的多个总线主设备是如何分享系统总线的？

13.5 MULTIBUS 系统总线支持哪两类中断处理方式？简述中断过程。

13.6 多总线中从设备的设计应注意哪些问题？

第十四章　高性能的16位微处理器——80186/80188

§14.1　引　　言

随着集成电路技术的发展，在单块集成电路芯片上所能实现的集成度就越来越高。这样就有可能制造出在性能和功能上都更加高级更加完善的微处理器。Intel 80186 和 80286，就

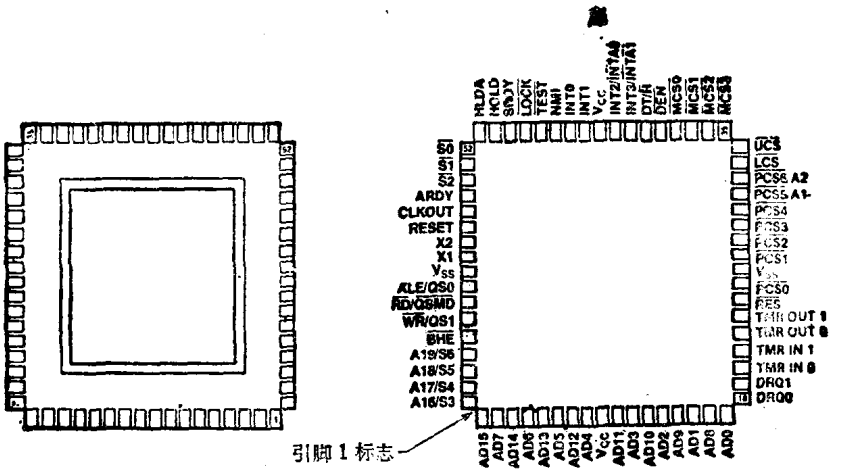


图 14.1A　80186 芯片引脚

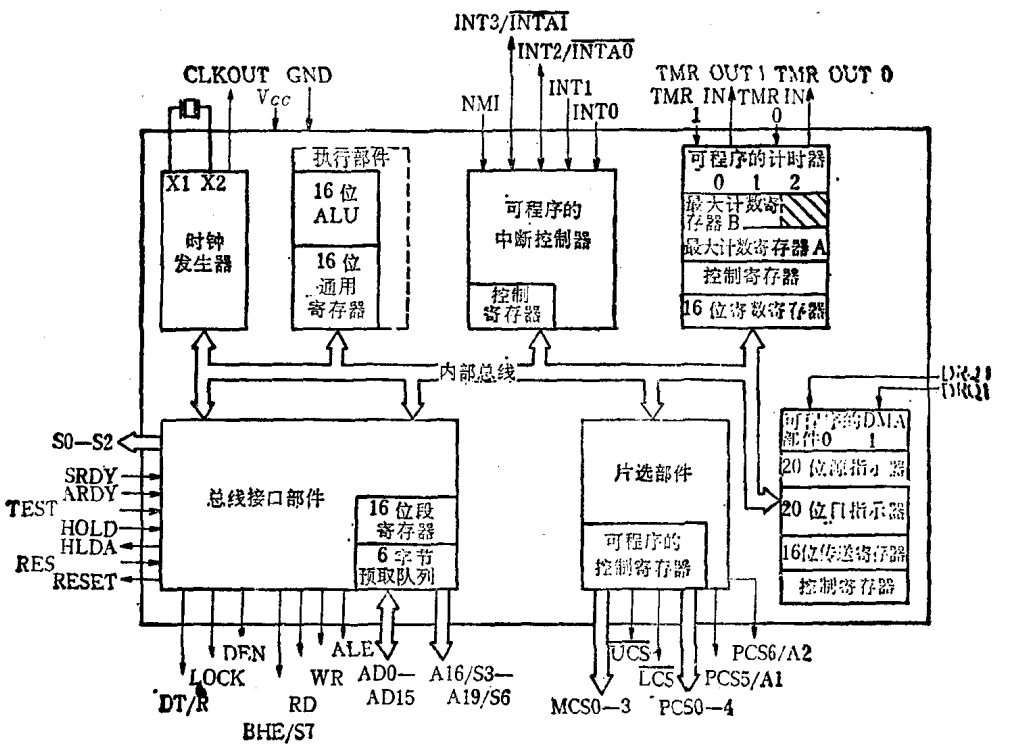


图 14.1B　80186 方框图

是集成电路技术发展到一个新高度的产物。

80186是一块具有68个引脚的芯片(见图14.1A)，在其中集成了六个功能块，而80286则在8086的基本结构上增加了存贮器保护和管理功能(关于80286的详细情况，可参见第十五章)。

本章的目的是想通过例子说明80186及各种外围和存贮设备的用途。由于80186把DMA部件，计时器部件，中断控制部件，总线控制部件，片选及准备就绪信号发生器和CPU集成在一块芯片上(见图14.1B)，系统结构就可以得到简化，因为许多原来要分别设计的接口现在已经集成在芯片上了。

80186系列实际上是由80186和80188两种微处理器组成。这两种微处理器的区别在于80186具有一条16位的外部数据总线，而80188则具有一条8位的外部数据总线。从内部来说，这两种处理器由相同的集成的外围器件所组成。因此，除了特别指出以外，本章中所叙述的关于80186的描述也同样适用于80188(§14.11.9中描述了80186和80188的区别)。本章所涉及的各种参数均是对8 MHz的80186而言的。

§14.2 80186概况

§14.2.1 CPU

80186 CPU具有一个与8086,8088,80286一样的公共的基本结构。它与8086/88是完全目标代码相兼容的。这个基本结构具有4个16位的通用寄存器(AX、BX、CX、DX)，这些寄存器可以作为8位或16位的操作数在大多数算术操作中使用。它还有4个16位的"指示器"寄存器(SI,DI,BP,SP)，这些寄存器既可以用在算术操作中，也可以用来对在存贮器中的变量进行访问。4个16位段寄存器(CS、DS、SS、ES)可以用来对存贮器进行简单的划分，以帮助构成程序模块。它还有一个16位的指令指示器和一个16位的状态寄存器。

80186产生物理存贮器地址的过程和8086相同，即把16位的段基地址值左移4位，然后和从由基地址寄存器、变址寄存器和位移量值的组合中导出的偏移量值相加(见图14.2)。这种加法的进位将被忽略，而这种加法的结果就是所需要的20位物理地址，它可以用来对系统存贮器寻址。

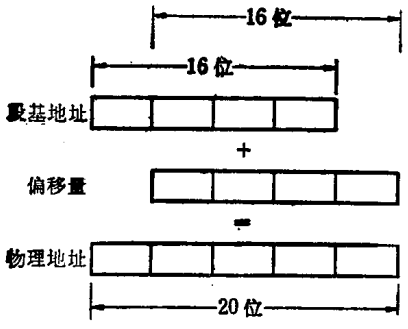


图14.2 在80186中物理地址的产生

80186有一个16位的ALU，它执行8位或16位的算术及逻辑操作。它为在寄存器、存贮器和I/O空间之间进行的数据移动提供了通路。CPU还可以使用串传送指令把数据从存贮器的一个区域高速传送到存贮器的另一个区域，或使用块I/O指令在I/O转接口和存贮器之间进行高速数据传送。此外，CPU还提供了许多条件转移和控制指令。

和在8086中一样，80186的取指令和指令的执行是由分开的总线接口部件和执行部件各自实现的。80186还有和8086一样的6字节指令预取队列。80188和8088一样，具有4字节的预取队列。在程序执行时，操作码由总线接口部件从存贮器中取出并放入这个队列。一旦执行部件需要另一条指令时，它就从该指令队列中去取。加了这个队列以后，由于总线接口部

件能在执行部件执行一条较长指令时连续地取指令，处理器的有效吞吐率就能得到提高。因为当处理器执行完一条指令时，它就再也不必等待下一条指令从存贮器中取出来，而只要到队列中去取就可以了。

§14.2.2 得到增强的 80186 CPU

虽然 80186 与 8086 是完全目标代码相兼容的，但大多数8086的指令在 80186 上执行起来却要比在 8086 上快，这是因为80186的总线接口部件和执行部件得到了硬件加强的缘故。例如，8086 是用内部微程序来计算地址(段基地址＋基地址＋变址＋位移量)的，而 80186 则用专用的硬件加法器来实现这个地址计算。80186 还包含了能把 8 MHz 的 8086 的 16 位整数乘除指令加速 3 倍执行的硬件。另外，通过流水线工作方式把递减 CX 和检查标志并行进行操作，从而使串处理的速度和存贮器能处理它们的速度一样快(可高达 2 兆字节/秒)。多位移位和环移执行每次移位的速度可达 1 位/时钟周期。此外，80186 还提供了许多新的指令，它们能简化汇编语言的程序编制，增强实现高级语言的性能和减少目标代码的长度(这些新的指令也包含在 80286 中)。

§14.2.3 DMA 部 件

80186 包含了一个 DMA 部件，该部件提供了两个高速的 DMA 通道。这个 DMA 部件能以字节或字为单位对 I/O 空间和存贮器空间的任意组合实现双向的传送。每个 DMA 周期需要二到四个总线周期，其中一个或两个总线周期用来把数据取到一个内部寄存器，一个或两个总线周期用来发送数据。这样就能够把字数据放在奇地址单元中，或把字节数据从奇地址单元传送到偶地址单元。通常，要这样做是比较困难的，因为奇地址字节是在 16 位数据总线的高 8 位传送的，而偶地址字节是在该数据总线的低 8 位传送的。

每个 DMA 通道都有独立的 20 位源和目指示器，它们用来访问被传送数据的源和目。这些指示器中的每一个都能独立地寻址 I/O 或存贮器空间。在每个 DMA 周期之后，这些寄存器能够独立地递增，递减或保持不变，每个 DMA 通道还保持有一个传送计数，它在到达预先编程的传送次数以后，就终止一个 DMA 进程。

§14.2.4 计 时 器

80186 含有一个计时器部件。该部件由 3 个独立的 16 位计时器/计数器组成。这些计时器中的两个能用来计数外部事件，提供从具有任何占空周期的 CPU 时钟或外部时钟中导出的波形，或在计时器的指定次数的“事件”之后中断 CPU。第三个计时器只计算 CPU 时钟并且可以用来在达到 CPU 时钟的可程序的周期数以后中断 CPU，以便在一个可程序的 CPU 时钟周期数以后向其他两个计时器分别或共同输出一个计数脉冲，或在一个可程序的 CPU 时钟计数之后给集成的 DMA 部件以一个 DMA 请求脉冲。

§14.2.5 中断控制器

80186 包含了一个中断控制器。这个控制器仲裁在所有内部和外部源之间的中断请求。它可以作为主控制器和两个外部的 8259A 中断控制器直接级联。此外，它也能作为一个外部中断控制器的从中断控制器，以便能实现与 80130，80150 以及 iRMX 86 操作系统的完全兼容。

§14.2.6 时钟发生器

80186 包含了一个时钟发生器和晶体振荡器。该晶体振荡器的时钟频率是所要求的 CPU 时钟频率的 2 倍(即对 8 MHz 的 80186 来说晶体振荡器的频率应该是 16 MHz)。如果使用外接的振荡器，它的频率也要 2 倍于 CPU 的时钟频率。这个振荡器的输出在内部被除以 2 以提供 CPU 时钟，根据这个 CPU 时钟再导出所有 80186 的系统定时。这种 CPU 时钟是可供外部使用的，所有的定时参数都是以这个外部可用信号为标准的。时钟发生器还为处理器提供了准备就绪同步信号。

§14.2.7 片选和准备就绪信号发生部件

80186 包含了集成的片选逻辑，它可以用来选择存贮器或外围设备。六条输出线可以用来作存贮器寻址，七条输出线可以用来作外围设备寻址。

为了能分离地寻址在一个典型的 80186 系统中的主存贮器区域，存贮器片选线被分成 3 组，分别用于在存贮器最高地址的启动 ROM 区，在存贮器最低地址的中断向量区以及在存贮器中部的程序存贮区。每个区域的容量是用户可程序的。最低存贮器单元和最高存贮器单元的地址各自固定在 00000H 和 0FFFFFH。中部存贮器的起始单元是用户可程序的。

七条外围选择线中的每一条寻址高于一个可程序的基地址的七个相邻的 128 字节的块。这个基地址可以放在存贮器空间中，也可以放在 I/O 空间中，以便实现外围设备的 I/O 或存贮器映象。

每一个编程的片选区域与一组可程序的准备就绪位相联系。这些准备就绪位控制一个集成的等待状态发生器。这样，就能在对与该片选区域相关的存贮器区域进行访问时自动地插入数个可程序的等待状态(0 到 3)。另外，每组准备就绪位中包括一个决定外部准备就绪信号(ARDY 和 SRDY) 是将被使用还是将被忽略(即尽管在外部引脚上还没有返回一个准备就绪信号，总线周期也将终止)的准备就绪位。总共有 5 个准备就绪位集，它们各自能为最高存贮区、最低存贮区、中部存贮区、外围设备 0～3 和外围设备 4～6 产生独立的准备就绪信号。

§14.2.8 集成的外围电路访问

集成的外围和片选电路是由用标准的输入访问的 16 位寄存器集控制的。这些外围控制寄存器都放在一个 256 个字节的块中，该块可以放在存贮器空间中，也可以放在 I/O 空间中。

由于对它们的访问就和访问外部设备一样，所以对集成的外围电路的访问和控制并不需要新的指令。

§14.3 80186的使用

§14.3.1 总线与80186的接口

一、概述

80186的总线结构与8086的总线结构是很相似的。它包括有一条多重地址/数据总线，还有各种控制和状态线（见表14.1）。每个总线周期至少需要4个CPU时钟周期，为了和外部的存贮器或外围设备的速度相适应，还可以加上必需的任意数量的等待周期。由80186 CPU启动的总线周期和由80186集成的DMA部件启动的总线周期一样。

表14.1 80186总线信号

功　　能	信　号　名
地址/数据	$AD_0 \sim AD_{15}$
地址/状态	$A_{16}/S_3 \sim A_{19}/S_6$, $\overline{BHE}/S_7$
多处理器控制	$\overline{TEST}$
局部总线仲裁	HOLD HLDA
局部总线控制	ALE, $\overline{RD}$, $\overline{WR}$, DT/$\overline{R}$, $\overline{DEN}$
多主设备总线	$\overline{LOCK}$
准备就绪(等待)接口	SRDY, ARDY
状态信息	$\overline{S_0} \sim \overline{S_2}$

在下面的讨论中所给出的定时值都是关于8 MHz的80186的。其他速度的80186的各种参数值将和这里叙述的不一样。

每一个80186总线周期的时钟周期被称为一个"T"状态，并且被顺序编号为T_1、T_2、T_3、T_W和T_4。在处理器不需要总线操作时（取指令，存贮器写，I/O读等），在T_4和T_1之间可以加入空闲T状态（T_I）。准备就绪信号控制要插入每个总线周期的等待状态（T_W）数。这个数可以在0到无穷之间变化。

一个T状态的开始是由CPU时钟的从高电平到低电平的转换标志的。每个T状态分成2个节拍，节拍1（或低电平节拍）和节拍2（或高电平节拍），它们各自出现在CPU时钟的低电平和高电平期间。图14.3表示了这种情况。

所有T状态的不同类型的总线活动可以参见图14.4。地址信号在T_1期间发出，数据信号在T_2、T_3、T_W和T_4期间发出。总线周期的开始是由处理器的状态信号在T_1状态之前的T状态（T_4或T_I）的中间从被动状态（全部为高电平）变成主动状态来标志的。由于在总线周期的第一个T状态之前的T状态期间要出现即将来临的总线周期的信息，所以就要产生两种不同类型的T_4或T_I：一种是T状态之后立即跟着一个总线周期，另一种是T状态之后立即跟着一个空闲T状态。

在第一种类型的T_4或T_I期间，关于即将来临的总线周期的状态信息就直接产生。这个信息在该T状态中间的80186时钟的低电平到高电平的转换之后不迟于t_{CHSV}（55ns）就可供使用。在第二种类型的T_4或T_I期间，状态输出保持不活跃（高电平），因为没有要马上开始的

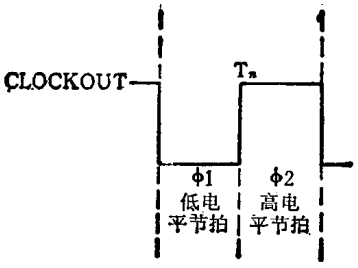


图 14.3 80186 中的 T 状态

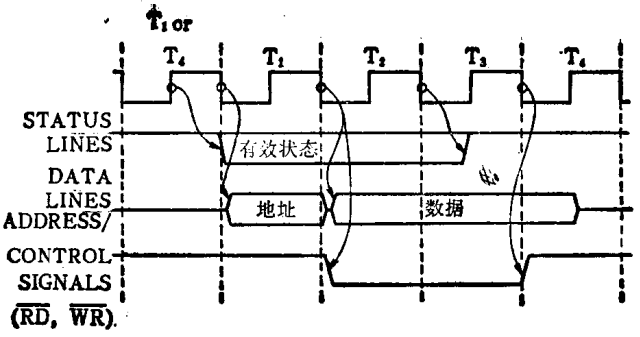


图 14.4 80186 总线周期举例

总线周期。这意味着每个 T_4 或 T_i 状态的性质是在 T_4 或 T_i 状态之前的一个 T 状态开始时决定的(见图 14.5)。这样做的结果就出现了总线等待时间。

二、物理地址的产生

物理地址是由 80186 在总线周期的 T_1 期间产生的。由于地址和数据线是公用的，所以这些地址若要在总线周期中保持稳定，就必须在 T_1 期间把它们锁存起来。为了实现锁存物理地址，80186 产生了一个高电平有效的 ALE(Address Latch Enable) 信号，它可以直接连接到一个锁存器的选通输入端。

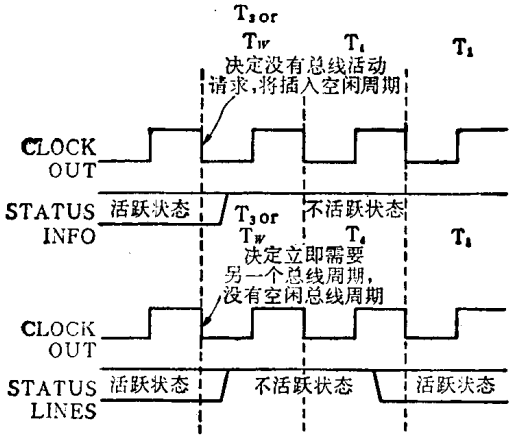


图 14.5 80186 中总线活跃——非活跃状态转换

图 14.6 说明了 80186 物理地址产生的参数。在 T_1 开始之后的 t_{CLAV}(44ns) 之内地址变成有效并在 T_1 结束之后保持有效至少 t_{CLAX}(10ns)。在 T_1 之前的 T 状态 (T_4 或 T_i) 的中间，ALE 信号被驱动为高电平，并在 T_1 的中间，在地址变为有效之后，一到 t_{AVAL}(30 ns) 就被驱动为低电平。这个参数 (t_{AVAL}) 要满足从地址有效到选通不活跃的地址锁存建立时间。在 ALE 不活跃以后，地址在地址/数据总线上至少稳定 t_{LLAX}(30ns) 以便满足从选通不活跃到地址失效的地址锁存保持时间。

由于 ALE 在地址变有效以前就变成高电平，所以地址锁存过程的延迟，将主要是锁存的传播延迟，而不是锁存选通的延迟。对 Intel 8282 锁存器来说，这个参数是 t_{IVOV}，即当选通保持活跃(高电平)时从输入有效到输出有效的延迟。注意，80186 比 8086、8288 总线控制器把 ALE 驱动为高电平早一个时钟节拍，并在 8086 或 8288 ALE 高电平

注：

1. t_{CHLH}：时钟高电平到 ALE 高电平 max=35ns
2. t_{CLAV}：时钟低电平到地址有效 max=44ns
3. t_{CHLL}：时钟高电平到 ALE 低电平 max=35ns
4. t_{CLAX}：时钟低电平到地址失效(地址从时钟低电平开始保持)=10ns
5. t_{LLAX}：ALE 低电平到地址失效(地址从 ALE 开始保持)min=30ns
6. t_{AVAL}：地址有效到 ALE 电平(地址建立到 ALE)min=30ns

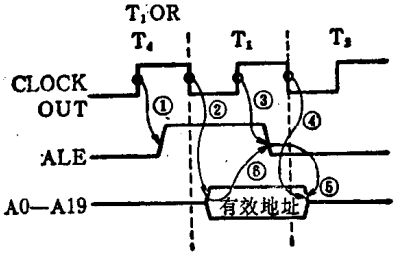


图 14.6 80186 地址产生定时

的时间内保持 ALE 为高电平(即 80186 的 ALE 脉冲比较宽)。

图 14.7 表示了一个典型的锁存物理地址的电路。这个电路使用了 3 个 8282 锁存器来锁存 80186 提供的 20 位地址。这些锁存器中的最高 4 位只用来选择各种存贮器部件或子系统，所以，当集成的片选使用时，这些最高位不需要锁存。该电路从 T_1 开始(包括 Intel 8282的地址锁存传播时间(t_{IVOV}))的最坏情况下地址产生的时间是：

$$t_{CLAV}(44\,ns) + t_{IVOV}(30\,ns) = 74\,ns$$

许多存贮器或外围设备并不需要在数据传送的整个期间保持地址稳定，例如 80130 和 80150 操作系统固件片和 21868 K×8 iRAM。若一个系统完全由这类设备组成，地址就不需要锁存。另外，80186 的两个片选输出可以组合来为在一个地址/数据总线不分路的系统中的外围寄存器的选择提供被锁存的 A_1 和 A_2 输出。

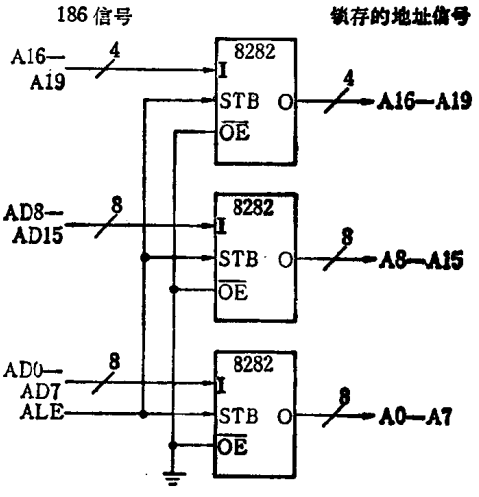


图 14.7 80186 的多路化的地址总线

80186 还产生了一个寻址存贮器的信号：$\overline{BHE}$ (Bus High Enable)。这个信号与 A_0 一起，使字节设备能连接到 16 位数据总线的高 8 位或低 8 位。由于 A_0 仅用来使设备能连接到数据总线的低 8 位，所以存贮器片选地址通常由地址位 A_1—A_{19} 来驱动，而不是由 A_0—A_{19} 来驱动。这样就提供了 512 K 字地址或 1M 字节地址。

当然，$\overline{BHE}$ 不出现在 8 位的 80188 中，在 80188 中所有的数据传送都在数据总线的 8 位上进行。

三、数据总线操作

在一个总线周期的 T_2、T_3、T_W 和 T_4 期间，多重地址/数据总线变成了 16 位的数据总线。在这条总线上，数据可以以字或字节传送。所有的存贮器是字节可寻址的，即一个 16 位字的较高字节和较低字节都有它们唯一的字节地址，用这个地址，它们可以被独立地访问。图 14.8 表示了具有同一字地址的两个字节分别有它们的字节地址的情形。

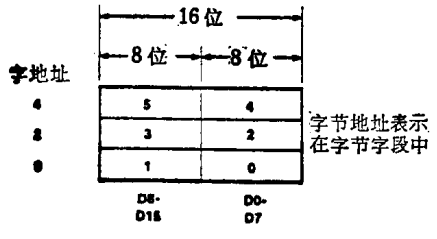


图 14 8 在 80186 中物理存贮器的字节/字寻址

所有具有偶地址 ($A_0 = 0$) 的字节常驻在数据总线的低 8 位，而所有具有奇地址 ($A_0 = 1$) 的字节常驻在数据总线的高 8 位。只要是对偶字节的访问 A_0 就是低电平，而 $\overline{BHE}$ 是高电平，并且数据传送是在 D_0—D_7 上进行。如果对奇字节访问，$\overline{BHE}$ 就是低电平而 A_0 是高电平，并且数据传送是在 D_8—D_{15} 上进行。如果对一个偶地址进行字访问，则 A_0 和 $\overline{BHE}$ 都被驱动为低电平并且数据传送在 D_0—D_{15} 上进行。

若对一个奇地址执行字访问，则要执行两次字节访问，第一次在 D_8—D_{15} 上访问第一个字地址的奇字节，第二次在 D_0—D_7 上访问下一个顺序字地址的偶字节。例如，在图 14.8 中，字节 0 和字节 1 可以用字地址 0 在两个分离的总线周期中分别进行访问(读或写)。它们也能够用字地址 0 在一个总线周期中进行访问。但是如果对地址 1 进行字访问，则需要两个总线周

期，第一个总线周期在字地址 0 访问字节 1（注意，字节 0 将不被访问），然后，在第二个总线周期中，在字地址 2 访问字节 2（注意，字节 3 将不被访问）。这就是若要提高处理器性能，就要把字数据放在偶地址的原因。

四、80188 数据总线操作

由于 80188 在外部只有一条 8 位数据总线，上面关于数据总线的高字节和低字节的讨论就不适用于 80188。若在存贮器空间中字数据放在偶地址也不会使性能改善，所有的字访问都需要两个总线周期，第一个访问该字的较低字节，第二个访问该字的较高字节。

80188 对集成的外围电路的访问必须按 16 位来进行，也就是说，要在一个总线周期中进行一个字访问，这是通过内部的 16 位总线进行的。80188 的外部数据总线只能传送一个字节。

五、通用数据总线操作

根据 80186 的总线驱动能力，在许多小系统中，该总线不需要附加的驱动。若在系统中没有使用数据缓冲器，就要注意避免在 80186 和直接连接到 80186 数据总线的设备之间出现争用总线的情况。由于 80186 在激活任何命令信号之前是浮空它的地址/数据总线的，所以在 80186 开始为下一个总线周期驱动地址信息以前，只要直接连接的设备在一次读以后浮空它的输出驱动器就可以了。这里有影响的参数是：从 $\overline{RD}$ 不活跃到下一个总线周期的地址活跃为止的最小时间（t_{RHAV}），这个时间的最小值是 85 ns。若存贮器和外围设备不能在这个时间里禁止它的输出驱动器，就需要数据缓冲器来避免 80186 和外围或存贮器设备同时驱动这些线。这个参数不受附加的等待状态的影响。数据缓冲器解决了冲突的问题，因为它们的把输出浮空的时间比 80186 所需要的最小时间要少得多。

在需要缓冲器的情况下，80186 提供了一个 $\overline{DEN}$ (Data Enable)和 DT/$\overline{R}$(Data Transmit/Receive)信号以简化缓冲器接口。$\overline{DEN}$ 和 DT/$\overline{R}$ 在所有总线周期期间不管该周期是否访问被缓冲的设备，总是会被激活的。一旦处理器准备好接收数据或处理器准备发送数据，$\overline{DEN}$ 信号就被驱动成低电平。在大多数系统中，$\overline{DEN}$ 信号应该不直接接到缓冲器的 $\overline{OE}$ 输入，因为不进行缓冲的设备(或其他的缓冲器)有可能直接连接到处理器的地址/数据引脚。若 $\overline{DEN}$ 被直接连接到若干个缓冲器，当许多设备试图驱动处理器总线时，在读周期期间将会发生冲

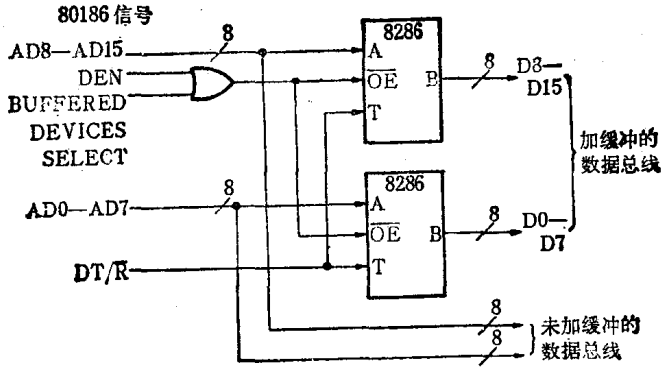


图 14.9　80186 缓冲的/不缓冲的数据总线的例子

突．另外，$\overline{DEN}$ 在产生允许双向缓冲器输入的输出信号时也应当（与被缓冲的设备的片选一起）成为一个因素。

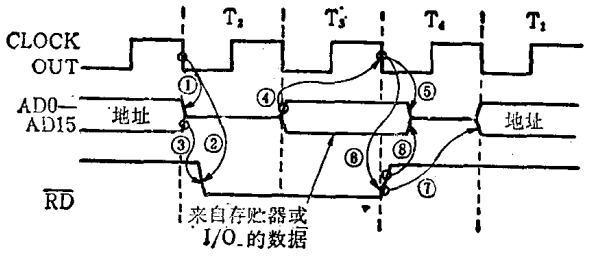
DT/$\overline{R}$ 信号决定通过双向总线缓冲器进行数据传送的方向。当数据是从处理器中被送出时，它是高电平，当数据是被读入处理器时，它是低电平。不象 $\overline{DEN}$ 信号，该信号可以直接连接到总线缓冲器，因为该信号通常并不直接启动缓冲器的输出驱动器。图 14.9 表示了一个既有缓冲的也有不缓冲的设备的数据总线子系统的例子。注意 8286 缓冲器的A边连接到80186，B边连接到外部设备。缓冲器的B边的驱动能力比A边大。DT/$\overline{R}$ 信号可以直接驱动缓冲器的 T(Transmit)信号，因为它具有对于这种结构的正确的极性。

六、控制信号

80186 直接提供控制信号 $\overline{RD}$，$\overline{WR}$，$\overline{LOCK}$ 和 $\overline{TEST}$。80186 还提供了状态信号 $\overline{S_0}$—$\overline{S_2}$ 和 S_6，从中可以产生所有其他必需的总线控制信号。

1. $\overline{RD}$ 和 $\overline{WR}$

$\overline{RD}$ 和 $\overline{WR}$ 分别把数据从存贮器或 I/O 空间中读入处理器，或把数据从处理器写向存贮器或 I/O 空间。$\overline{RD}$ 信号在整个存贮器和 I/O 读周期期间的 T_2 开始的时候被驱动为低电平，而在 T_4 开始的时候被驱动为高电平（见图 14.10）。$\overline{RD}$ 信号要在 80186 停止在地址/数据总线上驱动地址以后才变成有效。数据是在 T_4 开始的时候选通入处理器的，$\overline{RD}$ 要在处理器的数据保持时间(10ns)满足以后才变成不活跃。



1. t_{CLAZ}：时钟低电平到地址浮空 max=35ns
2. t_{CLRL}：时钟低电平到 $\overline{RD}$ 活跃 max=70ns
3. t_{AZRL}：地址浮空到 $\overline{RD}$ 活跃 min=0ns
4. t_{DVCL}：数据有效到时钟低电平（数据输入建立时间）min=20ns*
5. t_{CLDX}：时钟低电平到数据失效 min=10ns*
6. t_{CLRH}：时钟低电平到 $\overline{RD}$ 高电平 min=10ns
7. t_{RHAV}：RD 高电平到地址有效 min=85ns
8. t_{RHDX}：读高电平到数失据效 min=0ns*

* 80186 输入所需要的，其他的都是输出特性

图 14.10 80186 的读周期定时

注意，80186 并不提供分离的 I/O 和存贮器 $\overline{RD}$ 信号。若需要分离的 I/O 和存贮器读信号，则可以同步地使用 $\overline{S_2}$ 信号（低电平是所有的 I/O 操作，高电平是所有的存贮器操作）和 $\overline{RD}$ 信号（见图 11）。必须注意的是，若使用这种方法，$\overline{S_2}$ 信号就需要锁存，因为 $\overline{S_2}$ 信号（与 $\overline{S_0}$ 和 $\overline{S_1}$ 一样）在 T_4 开始之前就进入被动状态（$\overline{RD}$ 在那里也变成不活跃）。若 $\overline{S_2}$ 直接用于这个目的，读命令的类型（I/O 或存贮器）在 $\overline{S_2}$ 变成被动状态（高电平）的 T_4 之前的瞬间可能改变。状态信号可以用ALE信号象锁存地址一样锁存起来。

在 80186 系统中缺少分离的 I/O 和存贮器 $\overline{RD}$ 信号并不是一个关键问题，因为 80186 的每一个片选信号将只对应一种存贮器或 I/O 访问（存贮器片选只负责访问存贮器空间，而外围片选根据程序员的选择，既可以负责访问 I/O 空间也可以负责访问存贮器空间）。这样，片选信号仅在正确空间的正确地址上访问的时候才启动外部设备。

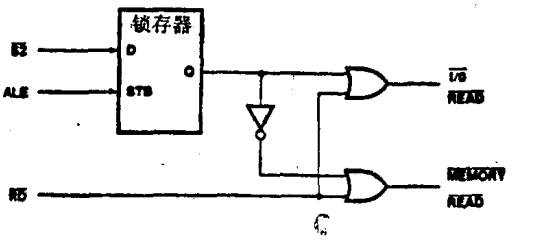
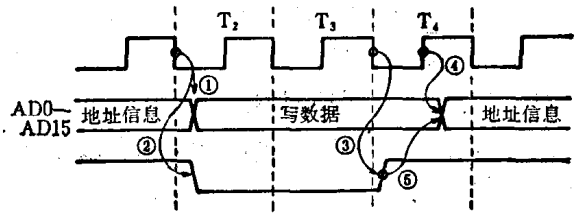


图 14.11 从 80186 中产生 I/O 和存贮器读信号

$\overline{WR}$ 信号也是在 T_2 开始时驱动为低电平，在 T_4 开始时驱动为高电平。象 $\overline{RD}$ 信号一样，$\overline{WR}$ 对所有的存贮器和 I/O 写都是有效的，并且分离的 I/O 和存贮器写信号也可以通过使用锁存的 $\overline{S_2}$ 信号和 $\overline{WR}$ 信号而得到(见图 14.12)。然而更重要的是，在 $\overline{WR}$ 进行它的活跃(高电平到低电平)变换时，有效的写数据并不出现在数据总线上。这样，当使用这个信号作为DRAM和 iRAM 的写启动信号时就会出现问题，因为这两种设备在 $\overline{WR}$ 信号的从不活跃转换的时刻，要求在数据总线上有稳定的写数据。在 DRAM 应用中，这个问题是用一个 DRAM 控制器(如 8207 或 8203) 来解决的，而对 iRAM，可以用在 CPU 和 iRAM 之间的 $\overline{WR}$ 线上放交叉耦合的与非门的办法来解决(见图 14.13) 这样将使 $\overline{WR}$ 信号到达 iRAM 延迟一个时钟节拍，以便能把有效的数据送到数据总线上去。



1. t_{CLDV}：时钟低电平到数据有效 max=44ns
2. t_{CVCTV}：时钟低电平到 $\overline{WR}$ 活跃 max=70ns
3. t_{CVCTX}：时钟低电平到 $\overline{WR}$ 不活跃 max=55ns
4. t_{CHDX}：时钟高电平到数据失效 min=10ns
5. $\overline{WR}$ 不活跃到数据失效 $=t_{CLCH}\min-t_{CVCTX}+t_{CHDX}$
$=55-55+10=10ns$

图 14.12 80186 的写周期定时

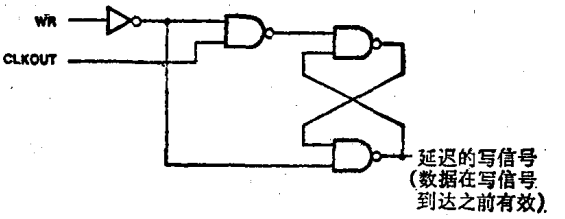


图 14.13 同步延迟来自 80186 的写信号

2．队列状态信号

若在启动和处理器操作期间，$\overline{RD}$ 线被外部接地，则 80186 将进入“队列状态”方式。在这种方式下，$\overline{WR}$ 和 ALE 信号就变成队列状态输出，反映在每一个时钟周期期间内部预取队列的状态。这些信号使协处理器(如 Intel 浮点处理器)能跟踪 80186 内部指令的执行情况。QS_0 (ALE)和 QS_1($\overline{WR}$)的解释给出在表 14.2 中。这些信号在时钟的高电平到低电平的转换时改变，比之在 8086 中早了一个时钟节拍。由于执行部件的操作是独立于总线接口部件操作的，所以队列状态线可以在任何T状态中改变。

表 14.2 80186 队列状态

QS_1	QS_0	解　　释
0	0	无操作
0	1	从队列中取出指令的第一个字节
1	0	队列被重新初始化
1	1	从队列中取出指令的后续字节

由于在 80186 构成队列状态方式时，ALE、$\overline{RD}$ 和 $\overline{WR}$ 信号在 80186 上不再可直接使用，所以这些信号必须使用一个外部的 8288 总线控制器从状态线 $\overline{S_0}$—$\overline{S_2}$ 中引出。为了防止 80186 在启动时偶然进入队列状态方式，在内部给 $\overline{RD}$ 提供了一个弱拉高器件。$\overline{RD}$ 是 80186 上仅有的三态的接有拉高或拉低器件的输入引脚。

3．状态信号

80186 提供了 3 种状态输出，它们用来指示当前正在执行的总线周期的类型。这些信号在一个总线周期的 T_1 之前的 T 状态期间(见图 14.5)从不活跃状态(全为高电平)变到七种可

表 14.3 80186 状态线解释

$\overline{S_0}$	S_1	$\overline{S_2}$	操　　作
0	0	0	中断响应
0	0	1	I/O 读
0	1	0	I/O 写
0	1	1	暂停
1	0	0	取指令
1	0	1	存贮器读
1	1	0	存贮器写
1	1	1	被动

能的活跃状态之一。状态信号可能的编码和它们的解释在表 14.3 中给出。这些状态信号，在当前总线周期的 T_4 之前的 T 状态中，被驱动到它们的不活跃状态。

状态信号可以直接连接到一个 8288 总线控制器，它可以用来提供局部总线的控制信号或多总线的控制信号（见图 14.14）。使用 8288 总线控制器并不妨碍使用 80186 产生的 $\overline{RD}$、$\overline{WR}$ 和 ALE 信号。例如，80186 直接产生的信号，可以用来提供局部总线控制信号，而 8288 则可用来提供多总线控制信号。

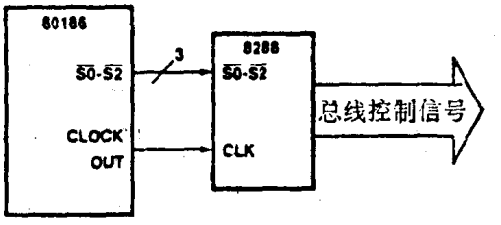


图 14.14 80186/8288 总线控制器相互连接

80186 提供了两个附加的状态信号：S_6 和 S_7。S_7 相等于 $\overline{BHE}$ 并和 $\overline{BHE}$ 共用同一引脚。$\overline{BHE}/S_7$ 在总线周期的 T_1 之前的 T 状态（T_4 或 T_i）的中间改变状态以反映即将执行的总线周期的性质。这就意味着 $\overline{BHE}/S_7$ 并不需要锁存，即它可以象 $\overline{BHE}$ 信号一样直接使用。S_6 提供了关于产生总线周期的部件的信息。它是与 A_{19} 分时的。并在 T_2，T_3，T_4 和 T_w 期间可供使用。在 8086 系列中，所有的中央处理器（如 8086，8088 和 8087 等）在它们的总线周期中把这条线驱动为低电平，而所有的 I/O 处理器（如 8089 等），则在它们的总线周期中把这条线驱动为高电平。根据这种模式，每当总线周期是由 80186 CPU 产生时，80186 就把这条线驱动为低电平，但当总线周期是由集成的 80186 DMA 部件产生时，就把这条线驱动为高电平。这样，外部设备就能够区分为 CPU 取数据和为 DMA 部件传送数据的总线周期。

80186 缺少 3 个 8086 可以使用的状态信号。它们是 S_3、S_4 和 S_5。原来 S_3 和 S_4 是用来表示所选择的段寄存器的，S_5 是用来表示中断触发器状态的。在 80186 中，这些信号总是低电平。

4. $\overline{TEST}$ 和 $\overline{LOCK}$

80186 还提供了一个 $\overline{TEST}$ 输入和一个 $\overline{LOCK}$ 输出。$\overline{TEST}$ 输入用来与处理器的 WAIT 指令相配合。它一般由一个协处理器（如 8087）驱动，以指示它是否正处于忙状态。在协处理器把 $\overline{TEST}$ 线驱动为低电平以指示它空闲以前，中央处理器可能会暂时被迫把所执行的程序挂起。

一旦执行封锁的指令的数据周期，$\overline{LOCK}$ 输出就被驱动为低电平。每当在一条指令的前面出现 LOCK 前缀，就产生了一条封锁的指令。LOCK 前缀只对紧跟在 LOCK 前缀后面的一条指令起作用。这个信号用来给总线仲裁器（如 8289）指出一系列封锁的数据传送正在进行，总线仲裁器在封锁的传送发生时，应该无条件地放弃总线。80186 将不再重新组织一个总线

HOLD，也不允许在封锁的数据传送期间，由集成的DMA控制器运行DMA周期。封锁的传送，用在多处理器系统的存贮器访问中，它起着一个控制对共享的系统资源进行访问的信号灯作用。

在80186中，$\overline{LOCK}$信号将在第一个封锁的数据传送周期的T_1期间变活跃，在最后一个封锁的数据传送周期开始以后，才被驱动为不活跃的3态。在8086中，$\overline{LOCK}$信号是在执行LOCK前缀之后立即被激活的。LOCK前缀可能在处理器准备执行封锁的数据传送的早得多的时间里执行，这样，就会产生在执行第一个封锁的数据周期之前就激活$\overline{LOCK}$信号的不良后果。由于$\overline{LOCK}$在处理器需要总线进行数据传送之前就被激活，就有可能封锁住了操作码的预取。然而，80186却能在处理器真正做好执行封锁传送之后，才激活$\overline{LOCK}$信号，所以80186不会发生封锁预取的情况。

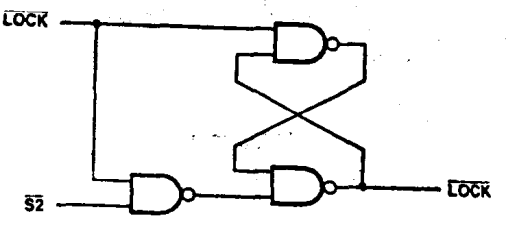


图14.15 保持$\overline{LOCK}$活跃直到返回RDY的电路

要注意的是：$\overline{LOCK}$信号并不一直保持活跃到最后一个封锁的数据传送周期结束，这一点在某些系统中可能会引起问题。例如，处理器请求从一个双向端口的RAM阵列中访问存贮器，并被拒绝进行立即访问(例如由于DRAM刷新周期)，而当处理器最终能够访问该RAM阵列时，它可能已经丢失了它的$\overline{LOCK}$信号，这样，双向端口控制器就有可能让其他的端口访问RAM阵列。图14.15举了一个电路的例子，该电路能够用来保持$\overline{LOCK}$活跃，直到80186接收到一个RDY信号。

七、暂停定时

HALT总线周期用来通知外部世界，80186 CPU已经执行了一条HLT指令。它在两个重要方面不同于普通的总线周期。

不同于普通的总线周期的第一个方面是：由于处理器进入了一个暂停状态，没有控制信号($\overline{RD}$或$\overline{WR}$)将会被驱动为活跃。处理器将不再驱动地址和数据信息，并且不会收到任何数据。第二个方面是：在该总线周期的T_2期间$\overline{S_0}$—$\overline{S_2}$将变成它们的被动状态（都是高电平）比之在普通的总线周期里变成被动状态要早得多。

ALE被驱动为活跃是和普通的总线周期一样的。由于没有表达有效的地址信息，所以被选通入地址锁存器的信息应该被忽略。但是，这个ALE脉冲，却可以用来锁存来自$\overline{S_0}$—$\overline{S_2}$状态线的HALT状态。

被暂停的处理器不和任何80186的集成的外围部件的操作冲突，这就意味着当处理器正处于暂停时，若要进行一个DMA传送，则将会执行DMA总线周期。事实上，在处理器处于暂停时，DMA的等待时间将会得到改善，因为DMA和处理器将不再在访问80186总线上发生冲突。

八、8288和8289接口

8288和8289是与8086和8088一起使用的总线控制器和多主设备总线仲裁器。由于80186的总线和8086的总线相类似，所以它们可以直接和80168一起使用。图14.16表示了一个80186和这两种器件相连接的情况。

8288总线控制器为8086的最大方式系统产生控制信号($\overline{RD}$、$\overline{WR}$、$\overline{ALE}$、DT/$\overline{R}$、$\overline{DEN}$

等)。它是通过对处理器的状态信号 $\overline{S_0}$—$\overline{S_2}$ 进行译码而导出这些信息的。由于 80186 和 8086 在这些状态线上驱动相同状态信息，所以 80186 可以和 8086 系统一样，直接和 8288 相连接。把 8288 和80186一起使用并不妨碍直接使用80186 控制信号，许多系统既需要局部总线控制信号又需要系统总线控制信号。在这种类型的系统中，80186 的控制信号线可以作为局部信号，而 8288 的控制信号线可以作为系统信号。要注意的是 8288 产生的 ALE 脉冲比 80186 本身的 ALE 信号出现得要晚一些。在许多多主设备系统中，8288 的 ALE 可以用来把地址选通入系统总线地址锁存器，以确保与地址保持时间相匹配。

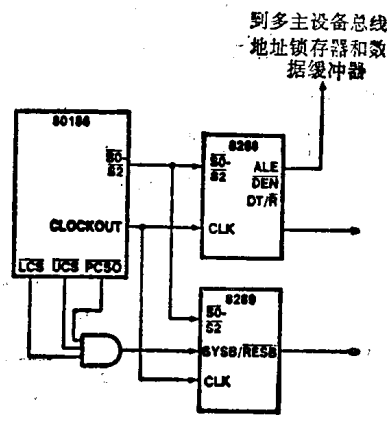
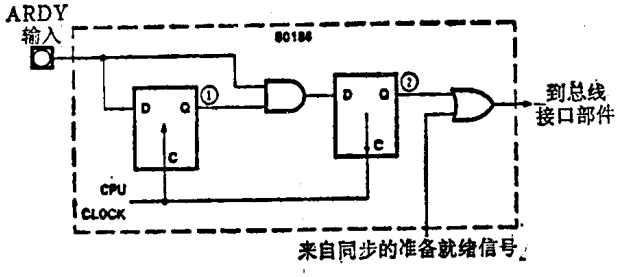


图 14.16　80186/8288/8289 的相互连接

8289 总线仲裁器在各种可以成为总线主设备的设备之间进行仲裁，以决定对多主设备系统总线的使用权。这个器件也对处理器的 $\overline{S_0}$—$\overline{S_2}$ 状态线进行译码，以便能直接决定何时需要系统总线。当需要系统总线时，8289 就强迫处理器等待，直到它获得对总线的控制为止，然后，它才允许处理器把地址、数据和控制信息驱动到系统总线上。系统通过地址译码，决定它何时需要系统总线资源。一旦被驱动的地址和一个在板资源的地址相符合，则系统总线就是不需要的，因此就不必再请求。图 14.16 所示的电路分解了 80186 的片选线以决定何时应当请求系统总线或 80186 的请求何时可以用一个局部资源来满足。

九、准备就绪接口

80186 提供了两种准备就绪引脚，一个是同步准备就绪 (SRDY) 引脚，另一个是异步准备就绪(ARDY)引脚。这些引脚上的信号通知处理器把等待状态 (T_W) 插入 CPU 总线周期。这样就使较慢的设备能响应 CPU 的服务请求(读或写)。等待状态将只在 ARDY 和 SRDY 都为低电平时才插入，即只要 ARDY 或 SRDY 信号中的一个活跃，就能终止一个总线周期。插入一个总线周期的等待状态的数目是任意的，在对集成的外围寄存器的任何访问期间，以及对由片选准备就绪位指示要忽略外部准备就绪信号的区域进行访问时，80186 将忽略 RDY 输入。

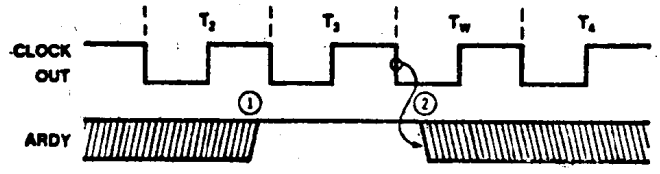


1. 异步消除触发器
2. 准备就绪信号锁存触发器

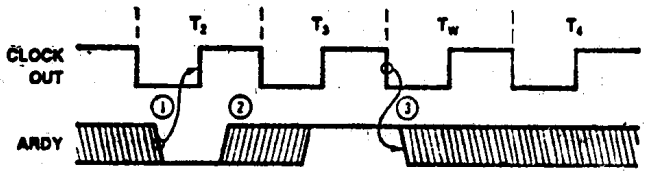
图 14.17　80186 的异步准备就绪电路

这两个 RDY 信号所需要的定时是不同的。ARDY 表示要使用异步的准备就绪输入，这样输入到这个引脚上的信号，将要先在内部和 CPU 的时钟同步，然后再送到处理器去。与 ARDY 引脚一起使用的同步电路在图 14.17 中表示。图 14.18 A 和图 14.18B 表示了 ARDY 信号有效和失效的变换(结果是插入等待状态)。第一个触发器用来"消除"ARDY 引脚上的异步的变化，在它的输出被锁存到第二个触发器并传送到 CPU 之前，它将取得一个确定的电平。当锁存的是高电平时，它就使在 ARDY 引脚上的电平直接传送到 CPU，当锁存的是低电平时，它就强制地把未准备就绪信号送给 CPU。

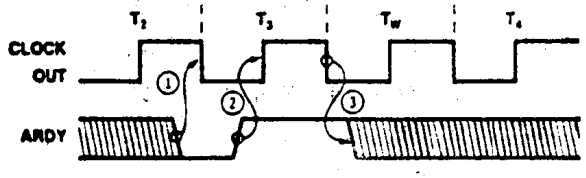
使用这种方式，只有 ARDY 信号的边沿是被同步的。一旦同步触发器被取样为高电平，ARDY 就直接驱动 RDY 触发器。由于这个 RDY 触发器的输入必须满足一定的建立和保持时间，因此信号向不活跃状态的变换满足这些建立和保持时间就很重要（t_{ARYLCV} = 35 ns、t_{CHARYX} = 15 ns）。ARDY 这样构成的原因，就是可使慢速的设备在被选中以后，有最大的时间来回答尚未准备就绪。在普通的准备就绪系统中，一个慢速设备（在它被选中以后，必须迅速地用尚未准备就绪来回答，以防止处理器继续从该慢速设备）上访问无效的数据。而使用上面的方法构成 ARDY，慢速设备就有了一个附加的时钟节拍，可以用来回答尚未准备就绪。



1. 不需要建立或保持时间
2. t_{CLARYX}：时钟低电平到 ARDY 不活跃（ARDY 活跃保持时间）min=15ns

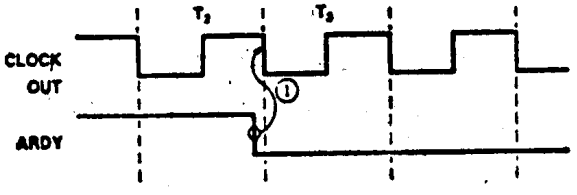


1. t_{ARYHCH}：ARDY 有效到时钟高电平（ARDY 不活跃建立时间到时钟高电平）min=20ns
2. 如果①能够保证，就不需要建立和保持时间
3. t_{CLARYX}：时钟低电平到 ARDY 不活跃（ARDY 活跃保持时间）min=15ns

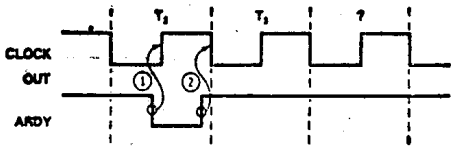


1. t_{ARYLCL}：ARDY 低电平到时钟低电平（ARDY 不活跃建立时间到时钟低电平）min=35ns
2. t_{ARYHCH}：ARDY 高电平到时钟高电平（ARDY 活跃建立时间）min=20ns（为了在下个时钟能够识别，这个时间必须被满足）
3. t_{CLARYX}：时钟低电平到 ARDY 不活跃（ARDY 活跃保持时间）=15ns

图 14.18A　有效的 ARDY 变换



① 小于 35ns



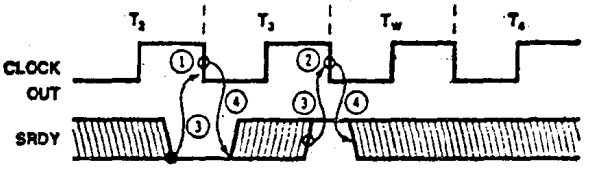
① 小于 20ns　② 小于 35ns

图 14.18B　无效的 ARDY 变换

若在 T_3 或 T_w 的开始时活跃的 RDY 被选通入 RDY 触发器（意味着在一个 T 状态的中间选通入同步触发器的 ARDY 是高电平，并且在下一个 T 状态开始之前保持高电平），则该 T 状

态之后将紧跟着一个 T_4。若在 T_3 或 T_W 的开始时低电平的 RDY 被选通入 RDY 触发器(意味着或是 ARDY 低电平选通入同步触发器或是 ARDY 高电平选通入同步触发器,但已在 ARDY 建立时间之前改变为低电平),则该 T 状态将紧跟着一个等待状态 (T_W)。在上述时间里,在 ARDY 引脚上不会出现任何异步变换,即当处理器不"检查"准备就绪信号时,不会引起 CPU 的错误动作。另外,ARDY 在插入等待状态时,SRDY 也必须驱动为低电平,因为它们是在内部相或来形成处理器 RDY 信号的。

同步的准备就绪(SRDY)信号,在 T_2,T_3,或 T_W 期间的所有变换,应该满足某个建立和保持时间 (t_{SRYCL} = 35 ns 和 t_{CLSRY} = 15 ns)。若这些要求不满足,CPU 就不能正确地操作。这个信号的有效变换和后续等待状态的插入表示在图 14.19 中。



1. 结果:未准备就绪,T 状态后将跟着一个等待状态
2. 结果:准备就绪,T 状态后将不跟一个等待状态
3. t_{SRYCL}:同步准备就绪稳定到时钟周期为低电平(SYDY 建立时间)min=35ns
4. t_{CLSRY}:时钟低电平到同步准备就绪转变(SRDY 保持时间)min=15ns

图 14.19 80186 的有效 SRDY 变换

处理器在每个 T_3 和 T_W 开始的时候检查这个信号。若在这两个状态的开始时检查到这个信号是活跃的,则该状态之后将紧跟着一个 T_4。否则,若在这两个状态开始时检查到该信号是不活跃的,则该状态之后将紧跟着一个 T_W。在 SRDY 引脚上的异步变换不在 T_3 或 T_W 开始时发生,即当处理器不"检查"准备就绪引脚上的信号时,就不会引起 CPU 的误动作。

十、总线特性总结

总线周期是连续出现的,但并不一定是一个紧接着一个的,也就是说,在由 80186 启动的每个总线访问之间,可以保持若干个空闲的 T 状态(T_i)。这种情况出现在 80186 内部的队列已满,并且没有由执行部件或集成的 DMA 部件提出读/写请求的情况下。读者可以回忆,总线接口部件从存贮器中取出操作码(包括立即数据),而执行部件实际上是执行预取出的指令。执行一条 80186 指令所需要的时钟周期数,可以从寄存器到寄存器传送的 2 个时钟周期一直到整数除法的 67 个时钟周期。

若一个程序含有许多长指令,程序的执行就会受到 CPU 的限制,即指令队列将总是充满的。这样,执行部件就不需要等待一条指令从存贮器中取出。若一个程序主要是含有短的指令或数据传送指令,它的执行就会受到总线的限制。在这种情况下,CPU 在继续它的操作之前,往往要等待指令从存贮器中取出来。这种附加等待状态使效率和性能降低的程序举例在 §14.11.7 中。

除了取指令操作是一个跳转到奇地址单元的结果以外,所有的取指令操作都是从偶地址中取一个字(16 位)。这样就最大限度地利用了每个总线周期来取指令,因为每次取指令将访问两个信息字节。

虽然总线的利用率,即 80186 用在取指令和执行上的时间的百分比因程序而异,但一条

典型的混合指令，在80186上执行比在8086上执行能获得更大的利用率。这是由高性能的执行部件以更大的速率在预取队列中取得指令而获得的。这也意味着等待状态的作用，在80186中比在8086中更加显著。但在大多数情况下，加等待状态使性能降低的影响可能比预计的要小，因为取指令和执行指令是由分离的部件进行的。

§14.3.2 存贮器系统举例

一、2764接口

有了上述的80186的知识以后，就可以构成各种存贮器接口。这些接口中最简单的EPROM接口，表示在图14.20中。

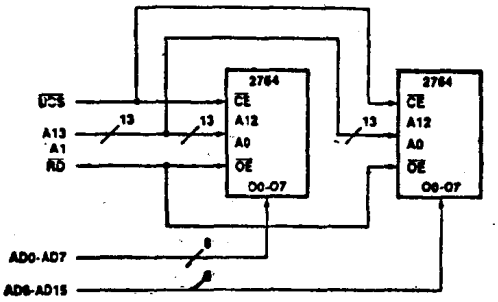


图14.20 2764/80186接口举例

地址用前面讲过的地址产生电路锁存。注意，每个EPROM的A0线是连接到来自80186的A1地址线而不是A0地址线。要记住，A0仅用来表示在16位数据总线的低8位上进行数据传送！EPROM的输出直接连接到80186的地址/数据输入端，并且80186的$\overline{RD}$信号用在EPROM的$\overline{OE}$上。

这种EPROM片子直接由80186的片选输出驱动。在这种结构中，关于EPROM访问时间的计算如下：

形成地址的时间：

$$(3+N)*t_{CLCL}-t_{CLAV}-t_{IVOV}(8282)-t_{DVCL}=375+(N*125)-44-30-20$$
$$=281+(N*125)\text{ns}$$

形成片选的时间：

$$(3+N)*t_{CLCL}-t_{CLCSV}-t_{DVCL}=375+(N*125)-66-20=289+(N*125)\text{ns}$$

形成$\overline{RD}$($\overline{OE}$)的时间：

$$(2+N)t_{CLCL}-t_{CLRL}-t_{DVCL}=250+(N*125)-70-20=160+(N*125)\text{ns}$$

这里：

t_{CLAV} = 从在T_1中的时钟低电平到地址有效的时间

t_{CLCL} = 处理器的时钟周期

t_{IVOV} = 从8282的输入有效到8282的输出有效

t_{DVCL} = 186数据有效输入建立时间到T_4的时钟低电平时间

t_{CLCLSV} = 从T_1的时钟低电平到片选有效

t_{CLRL} = 从T_2的时钟低电平到$\overline{RD}$变成低电平

N = 插入的等待状态数

这样，对0个等待状态的操作，EPROM必须使用250ns。上面没有计入的唯一重要的参数是t_{RHAV}，它是从$\overline{RD}$不活跃（高电平）到80186开始驱动地址信息的时间。这个参数是85ns，它与2764—25(250ns速度选择)的85ns的输出浮空时间相一致。若使用较慢的EPROM，则必须在EPROM数据线和地址/数据总线之间插入离散的数据缓冲器，因为这些器件在80186开始为下一个总线周期驱动地址信息的时候，可能会继续把数据信息送到多重地址/数据总线上去。

二、2186 接口

图 14.21 是一个 80186 和 2186iRAM 之间接口的例子。这个存贮器器件能很好地与 80186 匹配，因为它有较高的集成度，并且不需要地址锁存。

2186 是内部集成着刷新和控制电路的动态 RAM。它用脉冲方式和晚周期方式进行操作。进入脉冲方式，要求给该器件的 $\overline{CE}$ 是最多为 130ns 的低电平，并且要求命令输入($\overline{RD}$或 $\overline{WE}$)在 $\overline{CE}$ 之后的 90ns 内变成活跃。由于有这些要求，用脉冲方式把 80186 和 2186 接口将比较困难。为了取代这种方式，就使用了晚周期方式。这种方式提供了简单得多的接口，而又不使性能降低。根据控制信号的性质，iRAM 能自动地在这些方式之间选择。

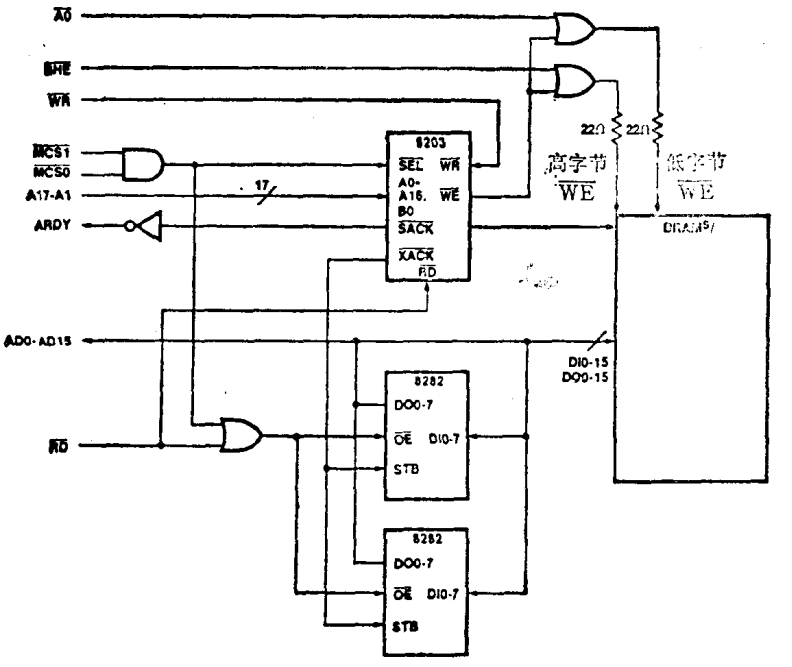


图 14.21 2183/80186 接口举例

2186 是一种主要的边沿触发器件。这就表示地址和数据信息是在命令信号的活跃变换(从高电平到低电平)时选通入该器件的。因此，要求 $\overline{CE}$ 和 $\overline{WR}$ 延迟到由 80186 驱动的地址和数据稳定为止。图 14.21 表示了一种能执行这种功能的简单电路。若地址没有被外部锁存，ALE 就不能用来延迟 $\overline{CE}$，因为这会破坏 2186 所需要的地址保持时间(30ns)。

由于 2186 是 RAM，所以数据总线允许信号 ($\overline{BHE}$ 和 A0) 就必须用来作为一个构成片允许信号，或 16 位 RAM 存贮器系统的较低和较高字节写允许信号的因素。若不这样做，则所有存贮器写，包括单字节的写都将写向该存贮器系统的较高和较低字节。在举例的系统中 $\overline{BHE}$ 和 A0 作为片选信号因素，分别连接到不同的 2186 $\overline{CE}$ 上。

2186 在它的片允许信号变为不活跃和片允许信号变为活跃之间需要一定的恢复时间，以确保能进行正确的操作。对一个"普通"周期 (一个读或写)，这个时间是 t_{EHEL} = 40ns。这表示 80186 的片选信号在一个总线周期的末尾将尽快地变得不活跃，以提供必须的恢复时间；即使是对该 iRAM 作连续的访问也是这样。若 2186 的 $\overline{CE}$ 信号被发出而缺少一个命令信号 ($\overline{WE}$ 或 $\overline{OE}$)，就会产生一个 FMC(False Memory Cycle)。一旦产生了一个 FMC，恢复时间

就要长得多，在 200ns 之内就不能启动另一个存贮器周期。因此，若存贮器系统将产生 FMC，则 $\overline{CE}$ 信号必须在紧靠在 T_4 之前的 T 状态(T_3 或 T_W)的中间除去，以确保对于 iRAM 的两个连续周期不会违反这个参数。状态变成被动(都是高电平)可以用于这个目的。这些信号在一个总线周期的最后一个 T 状态之前的 T 状态(T_3 或 T_W)的第一个节拍期间变成高电平。

由于 2186 是一个动态器件，它就需要刷新电路以保持数据的完整性。产生刷新周期的电路，是集成在 2186 中的。因此 2186 有一个准备就绪引脚，它用来在处理器 RAM 访问与内部产生的刷新周期冲突的时候，挂起处理器操作。这是一个开型的集合器输出，因而允许它们中的许多信号一起相"或"，因为在同一个时间可能访问多个器件。这些引脚通常是准备就绪的，也就是说只要 2186 不被访问，它们就是高电平。这些引脚只在处理器的请求和内部的刷新周期相冲突的时候，才被驱动为低电平。这样，在对 iRAM 进行访问的时候，来自 iRAM 的准备就绪信号必须成为输入 80186 RDY 电路的一个因素。由于 2186 刷新逻辑操作是异步于 80186 的，所以 RDY 信号为了与 80186 正确地一起操作就必须进行同步，这可以由集成的准备就绪同步电路或外部电路来做。举例的电路使用了与 ARDY 处理器输入相连的集成的同步电路。

除了处理器访问与内部的刷新周期相冲突，2186 的准备就绪信号总是活跃的。这些信号必须在一个请求把等待状态插入数据周期的周期之后尽快地变成不活跃。2186 在收到 $\overline{CE}$ 以后的 50ns 之内，把准备就绪信号驱动为低电平，这样在需要的时候，80186 就有足够的时间来插入等待状态。这里最关键的是：ARDY 在 T_2 中间的它的建立时间之前不被驱动成活跃，这是 80186 的异步准备就绪同步电路的特性所需要的。由于 2186 的准备就绪脉冲可以窄到 50 ns，所以，若准备就绪信号在同步电路的第一级就返回，并且在 T_2 末尾的 CPU 时钟高电平到低电平变换边沿的准备就绪信号的建立和保持时间中接着改变状态，则将产生错误的操作。

例子所示的接口，从 $\overline{CE}$ 开始具有 0 个等待状态的 RAM 读访问时间是：

$$3*t_{CLCL} - t_{CLCSV} - (\text{TTL 延迟}) - t_{DVCL} = 375 - 66 - 30 - 20\text{ns} = 259\text{ns}$$

这里：

t_{CLCL} = CPU 时钟周期时间

t_{CLCSV} = 从 T_1 中的时钟低电平到片选有效的时间

t_{DVCL} = 在 T_4 中的时钟低电平之前，80186 数据的建立时间

从 $\overline{OE}$ 活跃到数据有效延迟时间小于 100ns，因而它在这个接口中，不是一个访问时间的限制。另外，从 $\overline{RD}$ 不活跃到 2186 数据浮空时间，小于 80186 所需要的最大时间 85ns。图 14.21 中所示的 $\overline{CE}$ 产生电路提供了一个至少 11ns 的地址建立时间和一个至少为 35 ns 的地址保持时间(假定最大的两层 TTL 延迟小于 30 ns)。

写周期的地址建立和保持时间与读周期的时间相同。所示的电路，从 $\overline{WE}$ 开始提供了至少 11 ns 的数据建立和 100 ns 的数据保持时间，因此很容易满足 2186 需要的 0 ns 的建立和 40 ns 的保持时间。

三、8203 DRAM 接口

图 14.22 是一个 8203/DRAM 接口的例子。8203 提供了所有必须的 DRAM 控制信号、分路地址以及刷新操作。在这个电路中，8203 被构成与 64 K 的 DRAM 接口。

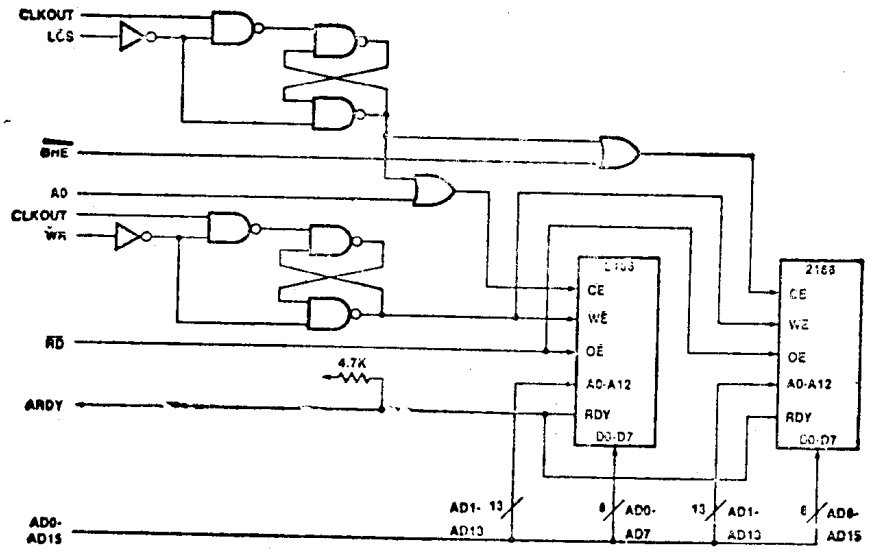
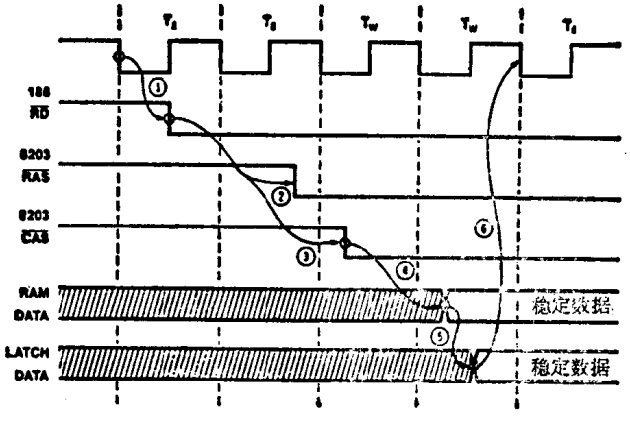


图 14.22　8203/DRAM/80186 接口举例

所有 8203 周期的产生，是与 80186 提供的控制信号(RD 和 WR)不相关的，这些信号要到总线周期的 T_2 才活跃。此外，由于 8203 的时钟(由 8203 内部的晶体振荡器产生)是异步于 80186 时钟的，所以，80186 提出的存贮器请求必须在该周期执行之前同步于 8203。为了减少进行这种同步所需要的时间，8203 应该使用能保持 DRAM 兼容性的最高速度的晶体。但即使使用 25MHz 的晶体(8203 所许可使用的最高频率)，例中所举的电路使用 150ns 的 DRAM 和 8 MHz 的 80186 时，还需要 2 个等待状态，若使用 200 ns 的 DRAM，则需要 3 个等待状态(定时分析见图 14.23)。



1. t_{CLRL}：时钟低电平到读低电平 max=70ns
2. t_{CL}：命令活跃到 RAS max=150ns*
3. t_{CC}：命令活跃到 CAS max=245ns*
4. t_{CAC}：从 CAS 开始的访问时间 max=85ns
5. t_{ISOU}：输入到输出的延迟 max=30ns
6. t_{DVCL}：数据有效到时钟低电平(数据建立)min=20ns

全部访问时间=70+245+85+30+20=450ns(3.6 个 T 状态)

①和⑥是 186 的说明　②和③是 8203 的说明　④是一个 2164A—15 的说明　⑤是关于 8282 的说明

* 假定是 25MHz 8203 的操作

图 14.23　8203/2164A—15 访问时间计算

由 8203 控制的整个 RAM 阵列，可由 80186 提供的一个或一组片选信号来选择，这些片选信号也可以用来插入该接口所需要的等待状态。

由于 8203 是异步于 80186 进行操作的，所以 8203 的 RDY 输出（在处理器的 DRAM 请求和 DRAM 刷新周期冲突的时候，用来挂起处理器操作）必须和 80186 同步。80186 ARDY 引脚用来提供必须的准备就绪同步信号。8203 的准备就绪输出用一种通常未准备就绪的方式进行操作，即仅在一个 8203 周期正在被执行、并且不处于一个刷新周期的时候，才被驱动为活跃，这和 2186 iRAM 所使用的通常准备就绪方式是根本不同的。8203 的 $\overline{\text{SACK}}$ 信号，仅在 DRAM 正被访问时才发送给 80186。要注意的是使用了 8203 的 $\overline{\text{SACK}}$ 输出，而不是 $\overline{\text{XACK}}$ 输出。由于在 RDY 被检查到为活跃和数据必须出现在数据总线的时间中，80186 将至少插入一个完整的 CPU 时钟周期，因此使用 $\overline{\text{XACK}}$ 信号，就会导致插入不必要的附加的等待状态，因为它要等到来自存贮器的有效数据可供使用，才指示准备就绪。

四、8207 DRAM 接口

8207 高级双向端口 DRAM 控制器，专门为 80186 或 80286 提供高性能的 DRAM 存贮器接口，即所有的分路地址和 DRAM 刷新电路。另外，它能从两个不同的端口中同步和仲裁存贮器请求（例如 80186 和一个 Multibus），并使两个端口共享存贮器。此外，8207 还提供了一个与检错和纠错片 8206 的简单接口。

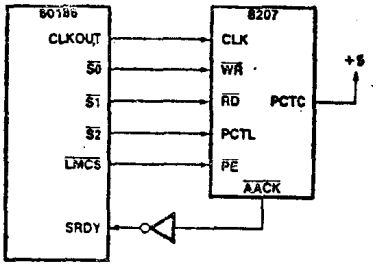


图 14.24　80186/8207/DRAM 接口

最简单的 8207（也是性能最高的）接口，表示在图 14.24 中。它说明了使用 8207 慢周期，同步状态接口把 80186 和 8207 相连接的情况。在这种方式中，8207 直接从来自 80186 的状态信号中，译码出要执行的周期类型。由于 8207 的 CLOCKIN 是由 80186 的 CLOCKOUT 驱动的，所以由因需要存贮器而请求在 80186 和 8207 之间同步，不会出现引起任何性能降低的情况。最后，和上述 8203 的情况一样，由 8207 驱动的整个存贮器阵列，都可以用 80186 的一个或一组存贮器片选信号来选择。

在上述接口中 8207 的 $\overline{\text{AACK}}$ 信号可以用来产生对 80186 的同步准备就绪信号。由于动态存贮器需要周期性的刷新，所以 80186 的访问周期，可能与 8207 产生的刷新周期产生冲突。当出现这种情况时，8207 将保持 $\overline{\text{AACK}}$ 引脚为高电平，直到处理器启动的访问能执行。这个信号应当成为 DRAM(8207) 选择输入的一个因素，并用来驱动 80186 的 SRDY 信号。注意，对一个要终止的总线周期来说，SRDY 和 ARDY 中只要有一个活跃就可以了。若异步设备（例如一个多总线接口）连接到 ARDY 引脚，而 8207 连接到 SRDY 引脚，在设计准备就绪电路时就要小心，使在同一个时间里只有一个 RDY 信号活跃，以避免出现总线周期的过早终止。

§14.3.3　HOLD/HLDA 接口

80186 采用 HOLD/HLDA 总线交换协议。这种协议使其他的异步总线主设备（即在总线上驱动地址、数据和控制信息的设备），能获得总线的控制权来执行总线周期（存贮器或 I/O 读或写）。

一、HOLD 响应

在 HOLD/HLDA 协议中，由正在请求总线控制权的设备（例如，外部 DMA 设备）升起 HOLD 信号，为了响应这个 HOLD 请求，80186 在完成它当前的总线活动以后，将升起它的 HLDA 信号。当外部设备完成总线操作以后，就降下它的总线 HOLD 请求。80186 就降下它的 HLDA 信号来响应，然后重新开始总线操作。

当 80186 通过把 HLDA 驱动为高电平，而响应一个总线保持请求时，它将浮空它的许多信号（见图 14.25）——AD_0-AD_{15}（地址/数据 0－15）和 $\overline{DEN}$（数据允许），在 HLDA 被驱动为活跃的时钟边沿以后的 t_{CLAZ}（35ns）之内浮空。$A_{16}-A_{19}$（地址 16－19）、$\overline{RD}$、$\overline{WR}$、$\overline{BHE}$、DT/$\overline{R}$ 和 $\overline{S_0}-\overline{S_2}$ 在紧靠使 HLDA 活跃的时钟边沿的前一个边沿之后的 t_{CHCZ}(45ns) 之内浮空。

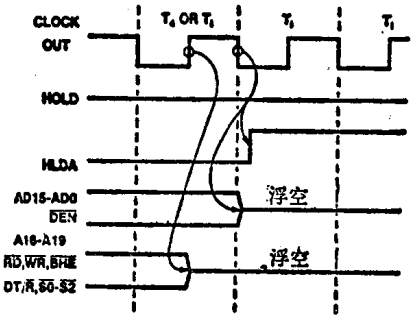


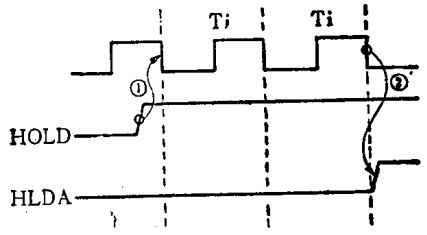
图 14.25　80186 的信号浮空/HLDA 定时

只有上面提到的信号，才是在总线 HOLD 期间浮空的，在不被 80186 浮空的信号中，有些是对执行外围功能有用的。其它的许多信号，则直接或间接地控制总线设备。这些信号是 ALE 和所有的片选信号（$\overline{UCS}$、$\overline{LCS}$、$\overline{MCS_{0-3}}$ 和 $\overline{PCS_{0-6}}$）。系统设计者必须知道：片选电路是不检查外部产生的地址的，这样，对由外部总线主设备寻址的存贮器或外围设备来说，就必须使用离散的片选和准备就绪发生电路。

二、HOLD/HLDA 定时和总线等待时间

从 HOLD 活跃到 80186 驱动 HLDA 活跃所需要的时间，称为总线等待时间。有许多因素影响这个等待时间，这些因素包括同步延迟、总线周期时间、封锁转移时间和中断响应周期。

HOLD 请求信号，是由 80186 内部进行同步的，因而是一个异步信号。为了确保能在某个时钟边沿识别，它必须满足对 CPU 时钟下降沿的一定的建立和保持时间。这种同步需要一个完整的 CPU 时钟周期，即 HOLD 信号在被从输入端锁存以后，要等一个完整的时钟周期，内部的 HOLD 信号才在内部总线仲裁电路上出现（参见 §14.11.2 关于 80186 同步电路的讨论）。若总线是空闲的，则 HLDA 将在 HOLD 之后两个时钟周期，加上少量的建立和传播延迟时间之后出现。第一个时钟周期同步输入，第二个时钟周期通知内部电路启动一个总线保持操作（见图 14.26）。



1. t_{HVCL}：保持有效到时钟低电平 min=25ns
2. t_{CLHAV}：时钟低电平到 HLDA 活跃 max=50ns

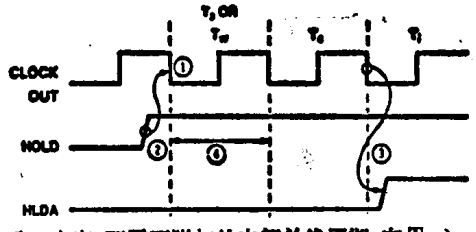
图 14.26　80186 空闲总线 HOLD/HLDA 定时

有许多因素影响着从一个 HOLD 请求，到一个 HLDA 之间的时钟周期数。从而使总线等待时间，比上述的最佳情况更长。最重要的因素可能是 80186 要到总线空闲，才肯放弃局部总线。只要 80186 不在执行任何总线传送，就会出现

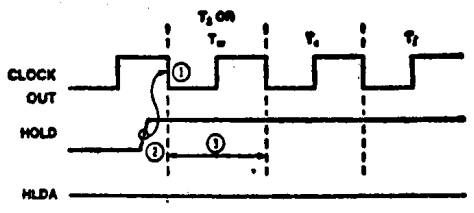
空闲总线。正如在§14.3.1中所说的那样，当总线空闲时，80186才产生空闲T状态。总线只能在一个总线周期的末尾变空闲，这样，80186只能在它当前总线周期的末尾之后，才能识别HOLD。80186在需要总线操作的时候，通常在T_4和下一总线周期的T_1之间是不插入T_i状态的。但是当80186在一个总线周期之后，不立即需要总线时，它将插入T_i状态，这种插入是独立于HOLD输入的。

当HOLD请求活跃时，80186将被迫从T_4进入T_i，以便释放总线。在总线周期结束之前，HOLD必须活跃3个T状态，以迫使80186在T_4之后插入空闲T状态(一个用来同步请求；一个用来通知80186，该总线周期T_4之后，将跟上空闲T状态)。在该总线周期结束之后，总线HOLD将被立即响应。若80186已经决定当前总线周期之后，将跟一个空闲T状态，则在总线周期结束以前，HOLD只需要活跃2个T状态，就能迫使80186在当前总线周期的末尾放弃总线。这是因为外部的HOLD请求，不必用来产生空闲的T状态。图14.27表示了上面所描述的情况。

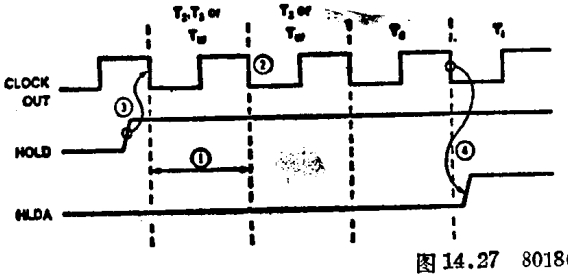
外部的HOLD，具有比80186 CPU和集成的DMA部件都高的优先级。但是，外部的HOLD，却不能把对在奇地址的字进行字访问的两个周期分开，也不能把使用DMA控制器的DMA传送的2—4个总线周期分开；每一个这样的因素，都将给80186的总线等待时间，加上一个附加的总线等待时间。



1. 决定：不需要附加的内部总线周期，在T_4之后将要插入空闲T状态
2. 大于25ns(t_{HVCL})
3. 小于50ns(t_{CLHAV})
4. 内部同步的HOLD请求



1. 决定：需要附加的总线周期，没有空闲周期将被插入；HOLD不能活跃得足够快来迫使T_4出现
2. 大于25ns(t_{HVCL})；因为它将不会被识别，所以不需要
3. 内部同步的HOLD请求



1. 内部同步的HOLD请求
2. 决定：HOLD请求活跃，空闲T状态将在当前总线的末尾插入
3. 大于25ns
4. 小于50ns

图14.27 80186的HOLD/HLDA定时

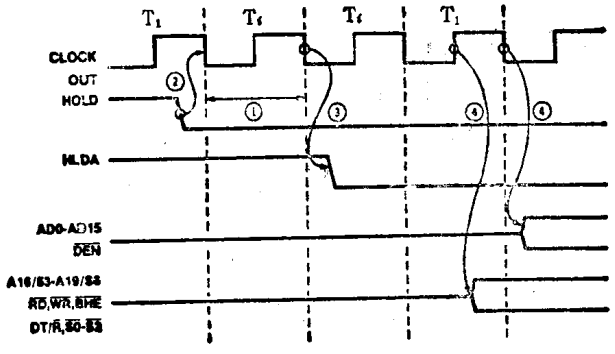
另一个影响总线等待时间的因素，是封锁的传送。一旦发生封锁的传送，80186就将不识别外部的HOLD(也不识别内部的DMA总线请求)。封锁的传送是由在指令前面的LOCK前缀决定的，任何由这样的加前缀的指令执行的传送，都将进行封锁并将不能被任何外部的总线请求所分离。串指令也可以被封锁，由于串传送可能需要上千个总线周期，因此，若它们被封锁，总线等待时间就会变得相当可观。

影响总线等待时间的最后一个因素，是中断响应周期。在使用一个外部中断控制器，或在iRMX 86方式下使用了集成的中断控制器，则80186将执行两个背靠背的中断响应周期。

这些周期是自动地被“封锁”的，故不管是内部的还是外部的总线 HOLD，都不能把它们分开。

三、退出 HOLD

在 80186 识别出 HOLD 信号已经变成不活跃以后，它就将在一个周期里，降下它的 HLDA 信号，图 14.28 表示了这种定时。若 80186 有内部总线周期要执行，80186 在 HLDA 变成不活跃以后将只插入 2 个 T_i。在后一个 T_i 期间，关于即将要执行的总线周期的状态信息将变活跃。若 80186 没有预期的总线活动，它就会使它的所有的信号线浮空，直到它开始 HOLD 以后的第一个总线周期之前的 T_i 为止。



1. 内部同步的 HOLD
2. 大于 25ns
3. 小于 50ns
4. 仅当一个总线周期要开始时，信号才退出浮空状态

图 14.28 80186 退出 HOLD

§14.3.4 8086 总线和 80186 总线的区别

80186 总线是与 8086 总线向上兼容的。因此 8086 总线接口器件 (8288 总线控制器和 8289 总线仲裁器)，可以直接在 80186 上使用。但是在这两种处理器之间，也有一些必须要考虑的区别：

1. CPU 占空周期和时钟发生器

80186 使用集成的时钟发生器，它提供了 50% 的 CPU 时钟占空周期 (1/2 的时间是高电平，1/2 的时间是低电平)。这一点和 8086 不同，8086 使用外部的时钟发生器 (8284A)，它具有 33% 的 CPU 时钟占空周期(1/3 的时间是高电平，2/3 的时间是低电平)。它们的这些不同可以表示如下：

(1) 在 80186 上没有振荡器输出，而在 8284A 时钟发生器上则相反。

(2) 80186 不提供 PCLK(50% 占空周期，1/2 CPU 频率)输出，而 8284A 则提供。

(3) 与相同速度的 8086 相比较，80186 的时钟低电平节拍较窄，而时钟的高电平节拍较宽。

(4) 80186 在内部不用 RDY 影响 ALE。这表示若两种 RDY 输入 (ARDY 和 SRDY) 都使用，则必须使用外部逻辑以防止对某个设备进行访问时，把没有连接到这个设备的 RDY 信号驱动为活跃。

(5) 80186 同时提供一个异步准备就绪输入和同步准备就绪输入，而 8284A 由用户选择提供两个同步准备就绪或两个异步准备就绪。

(6) 80186 的 CLOCKOUT(CPU 时钟输出信号)的驱动能力，小于 8234A CPU 时钟驱动能力。这表示能连接到这个信号上的高速设备，比能连接到 8284A 的时钟输出信号上的高速设备要少。

(7) 80186 使用的晶体或外部振荡器的频率，是 CPU 时钟频率的 2 倍，而 8284A 所使用的晶体或外部振荡器的频率，是 CPU 时钟频率的 3 倍。

2. 局部总线控制器和控制信号

80186 在使用 8288 总线控制器的情况下，同时提供局部总线控制器输出 ($\overline{RD}$、$\overline{WR}$、$\overline{ALE}$、$\overline{DEN}$ 和 DR/$\overline{R}$) 以及状态输出 ($\overline{S_0}$、$\overline{S_1}$、$\overline{S_2}$)，这是和 8086 不同的。在 8086 中若需要把状态输出(只在最大模式下产生)，就要牺牲局部总线控制器的输出(只在最小模式下产生)。在 80186 系统和 8086 系统之间的这些区别，可以说明如下：

(1) 由于 80186 能同时提供局部总线控制信号和状态输出，所以许多支持一个系统总线(如一个 Multibus) 和一个局部总线的系统就不再需要两个分离的外部总线控制器。这就是说，在 80186 的状态信号被连接到 8288 总线控制器去驱动系统总线的控制信号的同时，80186 的总线控制信号可以用来控制局部总线。

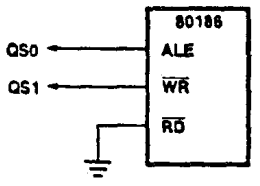
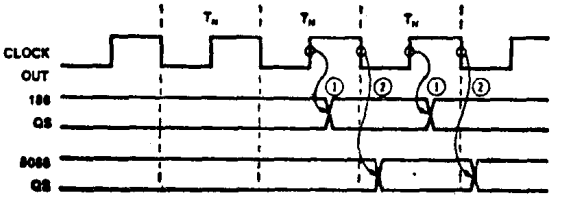


图 14.29 从 80186 产生队列状态信息

(2) 80186 的 ALE 信号，比 8086 或 8288 的 ALE 信号早一个时钟节拍，这样，就减少了地址锁存的地址传播时间。因为一般的从输入有效，到锁存的延迟时间，比从选通输入活跃开始的传播延迟时间要少。

(3) 80186 提供队列状态输出时，80186 的$\overline{RD}$必须接地(见图 14.29)。当这样进入"队列状态方式"以后，ALE 和 $\overline{WR}$ 输出就提供队列状态信息。注意，这个队列状态信息，比 8086 的要早一个时钟节拍可供使用(见图 14.30)。



1. 80186 在 CLK 的下降沿改变队列状态
2. 8086 在 CLK 的上升沿改变队列状态

图 14.30 80186 和 8086 队列状态的产生

3. HOLD/HLDA 和 RQ/GT

正如前面所讨论的那样，80186 使用 HOLD/HLDA 类型的协议，来改变总线的所有权(与 8086 最小方式一样)，而不是由 8086 在最大方式下使用的 RQ/GT 协议。这样就能与 Intel 的新一代的高性能/高集成度的外围总线主设备相兼容(例如 82586 Ethernet 控制器或 82730 高性能 CRT 控制器/文本协处理器)。

4. 状态信息

80186 不提供 S_3—S_5 状态信息。在 8086 中，S_3 和 S_4 用来提供产生当前正在执行的总线周期的物理地址的段寄存器的信息。S_5 用来提供关于中断允许触发器的信息，这些状态位在 80186 中总是低电平。

状态信号 S_6，被用来指示当前的总线周期是由 CPU 启动的还是由 DMA 设备启动的。结果，它在 8086 上总是低电平。在 80186 上，只要当前的总线周期是 80186 CPU 启动的，它就是低电平；而当前的总线周期是 80186 集成的 DMA 部件驱动的时候，它总是高电平。

5. 总线驱动

80186 输出驱动器，将驱动 200 pF 的负载，这是 8086(100pF)的 2 倍。这就使得在不用总

线缓冲器的情况下，也能构成较大的系统。它也意味着给 80186 提供好的接地是十分重要的，因为它的较大的驱动器能够很快地释放输出而在 80186 的接地引脚上产生很大的电流变化。

6．其他

80186 并不象 8288 总线控制器那样提供早的和晚的写信号。由 80186 产生的 $\overline{WR}$ 信号，相当于 8288 的早的写信号。它表示在这个信号被驱动为活跃时，地址/数据总线上的数据还没有稳定。

80186 也不提供不同的 I/O 和存贮器读和写命令信号，若需要这些信号，可以使用外部的 8288 总线控制器，或使用 $\overline{S_2}$ 信号来合成不同的命令。

§14.4 DMA 部件接口

80186 提供了一个含有 2 个独立的高速 DMA 通道的 DMA 部件。这些通道的操作是独立于 CPU 的，并且能象 CPU 一样驱动所有的集成的总线接口器件（总线控制器，片选等），图 14.31 表示了这种情况。在这里，由 DMA 部件启动的总线周期和由 CPU 启动的总线周期（除了在所有由 DMA 启动的总线周期期间 $S_6 = 1$ 以外）完全一样。这样，与 DMA 部件接口这件事本身就非常简单，因为除了附加的 DMA 请求连接以外，它和与 CPU 接口完全一样。

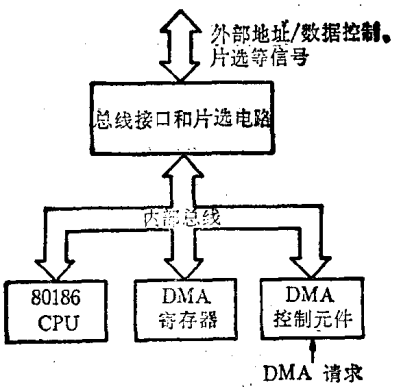


图 14.31 80186 CPU/DMA 通道内部模块

§14.4.1 DMA 的特性

两个 DMA 通道中的每一个，都具有如下的特性：

1．独立的 20 位源和目指示器，它们用来访问要从中取出数据，或把数据送到其中去的 I/O 或存贮器单元。

2．可程序的在每个 DMA 传送之后，对源或目指示器的自增和自减操作。

3．可程序的在某一数量的 DMA 传送之后，终止 DMA 操作。

4．可程序的在 DMA 终止时，中断 CPU。

5．对为偶或为奇的存贮器或 I/O 地址，进行字节或字的 DMA 传送。

6．DMA 的可程序产生的请求：

(1) 数据的源；
(2) 数据的目；
(3) 计时器 2；
(4) DMA 部件本身(继续 DMA 请求)。

§14.4.2 DMA 部件的编程

两个 DMA 通道中的每一个，都含有许多寄存器，通道的操作由这些寄存器来控制。这些寄存器包含在 80186 的集成的外围控制块中（参见§14.11.1）。这些寄存器包括源和目指示器寄存器，传送计数寄存器和控制寄存器。这些寄存器的位功能表示在图 14.32 中。

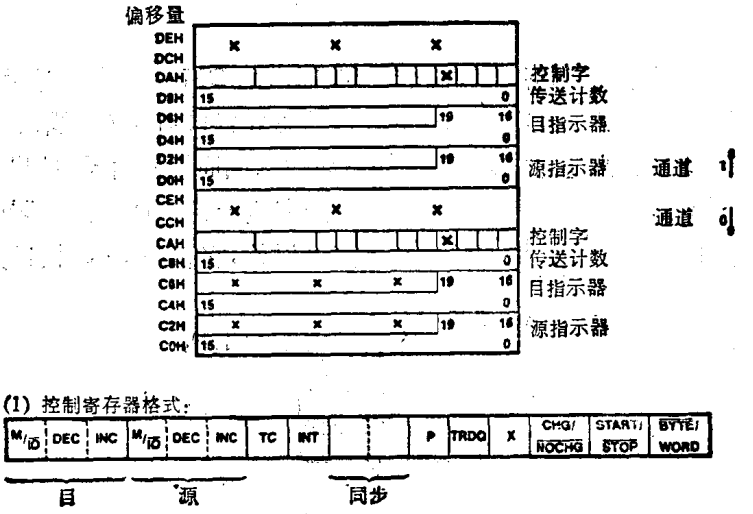


图 14.32 80186 DMA 寄存器的布局

20 位的源和目指示器，使得 DMA 部件对 80186 的全部 1 兆字节的地址空间，都能进行访问，并且所有的 20 位都受到 DMA 的自增、或自减部件的影响（即 DMA 通道把 80186 的 1 兆地址空间，看成平直的、不分段的线性数组来寻址）。在寻址 I/O 空间时，DMA 指示器寄存器的高 4 位被编程为 0。若它们不被编程为 0，则那些被编程的值(大于在 I/O 空间中的 64 K 空间)，将被驱动到地址总线上(一个对 CPU 来说不可访问的区域)。然而，即使在这种情况下，数据传送也能正确地进行。

在每次 DMA 传送之后，16 位的 DMA 传送寄存器被减 1，字传送和字节传送都一样。若在 DMA 控制寄存器中的 TC 位置位，则 DMA ST/$\overline{\text{STOP}}$ 位在计数寄存器变成 0 时将被清除，从而使所有的 DMA 操作停止。数值为 0 的传送计数，允许进行 65536(2^{16}) 次传送。

DMA 控制寄存器，含有控制各种通道特性的位(见图 14.33)，包括每个通道的数据源和目指示器是指向存贮器还是 I/O 空间，以及指示器在每次 DMA 传送之后是被递增、递减还是单独处理，它仍有在字节或字传送间进行选择的一个专用位。两个同步位用来决定 DMA 请求的源。在达到一个可编程的 DMA 传送数以后，TC 位决定 DMA 的操作是否要停止，INT 位用来在这种情况发生时允许中断处理器(注意，在 INT 位和 TC 位都置位时，才能产生对 CPU 的中断)。

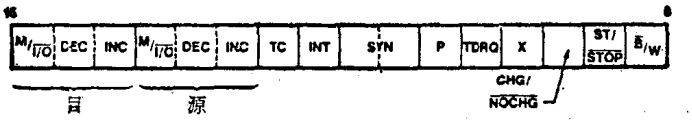


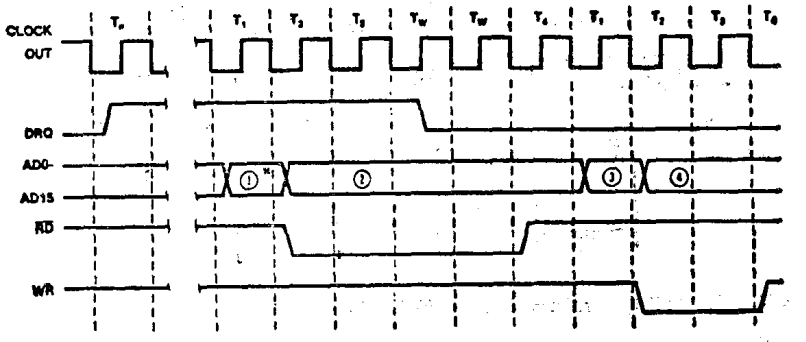
图 14.33 DMA 控制寄存器

控制寄存器还含有一个启动/停止位(ST/$\overline{\text{STOP}}$)，这一位用来启动 DMA 传送。一旦这一位置位，通道就“作好准备”，即只要对这个通道作出 DMA 请求，就会进行 DMA 传送。若这一位被清除，这个通道就不会执行 DMA 传送。与这个位成对的是 CHG/$\overline{\text{NOCHG}}$ 位，该位允许 DMA 控制寄存器的内容被改变，而不影响启动/停止位的状态。ST/$\overline{\text{STOP}}$ 位仅在写 DMA 控制寄存器期间且 CHG/$\overline{\text{NOCHG}}$ 也置位的时候，才能进行修改。CHG/$\overline{\text{NOCHG}}$ 位是只写的，它将总是被读出为 1。由于 DMA 传送可能在 ST/$\overline{\text{STOP}}$ 位置位以后马上发生，所以它应该在所有其他的 DMA 控制器寄存器被编程之后再置位。当传送计数寄存器到达 0，并且 DMA 控制寄存器中的 TC 位置位时，或当传送计数寄存器到达 0、并且非同步的 DMA 传送被编程时，该位就被自动地清除。

所有的 DMA 部件的可编程寄存器，都是 CPU 可直接访问的，这表示 CPU 能修改 DMA 部件的寄存器。例如，在进行了 137 次 DMA 传送之后，要以新的指示器值进行第 138 次 DMA 传送，就可以由 CPU 来修改源指示器寄存器。若在 DMA 通道里有多个寄存器在可能产生 DMA 请求并且该 DMA 通道是允许的任何时间要进行修改，寄存器的编程值可以放在存贮器单元里，然后使用一条封锁的串传送指令，把这些值送到 DMA 寄存器中去。这样能避免在只有一半寄存器值被改变之后，发生 DMA 传送。上述情况对于执行读/修改/写类型的操作也成立（例如，用一条 AND 指令“与”掉在指示器寄存器在存贮器空间的映象中的某些位）。

§14.4.3 DMA 传送

80186 中的每一个 DMA 传送，都由 2 个独立的总线周期，即取周期和发送周期所组成(见图 14.34)。在取周期期间，使用在源指示器寄存器中的地址访问存贮器、或 I/O 空间中的字节或字数据。取出的数据被放在一个内部的临时寄存器中，该寄存器是 CPU 不能访问的。



1. 源地址 2. 源数据 3. 目地址 4. 目数据

注：在总线周期期间，由总线状态而不是由 DMA 控制器插入等待状态

图 14.34 80186 DMA 传送周期的例子

在发送周期期间，就用在目指示器寄存器中的地址，把放在这个内部寄存器中的字节或字数据，传送到存贮器或 I/O 空间中去。这样的两个总线周期，将不能被总线 HOLD、或由其他的 DMA 通道所分离，除了在 CPU RESET 时之外，离开了一个周期，另一个周期就不能执行。由 DMA 部件执行的总线周期与由 CPU 执行的存贮器或 I/O 总线周期完全一样，这两种周期之间的唯一不同是 S_6 状态信号，在所有由 CPU 启动的总线周期中，该状态信号将被驱动为低电平，而在所有由 DMA 启动的总线周期中，该状态信号将被驱动为高电平。

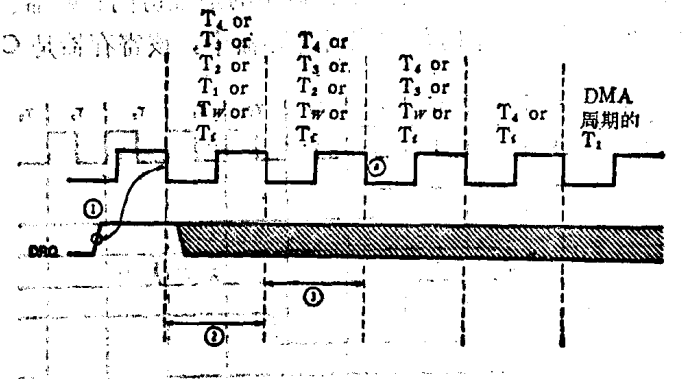
§14.4.4 DMA 请 求

每一个 DMA 通道有一条 DMA 请求线，外部设备通过这条线能够请求 DMA 传送。DMA 控制寄存器中的同步位，决定把这条线看作是连接到 DMA 数据的源、还是 DMA 数据的目。所有在这条线上的 DMA 请求，在被送到内部的 DMA 逻辑之前，要同步于 CPU 时钟。这表示 DMA 请求线上的任何异步变换，将不会引起 DMA 通道的错误动作。除了外部请求之外，DMA 请求，还可以由内部计时器 2 的超时，或对在 DMA 控制寄存器中的同步位连续地用程序设置为调用非同步的 DMA 传送而产生。

在产生任何 DMA 请求之前，80186 的内部总线必须转让给 DMA 部件。CPU 把这个内部总线转让给 DMA 部件，需要一定的时间。在发出 DMA 请求到执行 DMA 传送之间的时间，被称为 DMA 等待时间。关于 DMA 等待时间的许多细节和总线等待时间的细节相同，仅有的重要区别是，外部的 HOLD 的优先级比内部的 DMA 传送的优先级高。这样，在外部总线 HOLD 期间，内部 DMA 周期的等待时间就比较长。

每一个通道具有一个由程序指定的相对于另一个 DMA 通道的优先级。两个通道可以被程序指定为具有相同的优先级，或其中的一个通道被程序指定为比另一个通道具有更高的优先级。若两个通道都活跃，较低优先级的通道的 DMA 等待时间就比较长。若两个通道都活跃，并且两个通道具有相同的程序指定的优先级，则 DMA 传送周期将在两个通道之间交替（即，第一个通道执行取和发送之后，就进行第二个通道的取和发送）。

产生一个 DMA 周期的最小定时，表示在图 14.35 中。从 DRQ 变活跃，到第一个 DMA



1. t_{DRQOL}=DMA 请求到时钟低电平 min=25 ns，以保证识别
2. 同步电话分解时间
3. DMA 部件优先权仲裁等时间
4. 总线接口部件锁存 DMA 请求并决定执行 DMA 周期

图 14.35 80186 的 DMA 请求定时（所示的为对请求的最小响应时间）

周期开始的最小时间，是 4 个 CPU 时钟周期，这就是说，DMA 请求将在一个总线周期开始之前的 4 个时钟周期被取样，以决定是否需要 DMA 操作。这个时间是独立于插在总线周期中的等待状态的个数的，最大的 DMA 等待时间是其他处理器操作的函数。

若 DRQ 在图 14.35 中的 1 处被取样为活跃，则尽管 DMA 请求在第一个 DMA 周期开始之前就变得不活跃，该 DMA 周期还是要被执行。但这并不意味着 DMA 请求的被锁存入处理器会使任何在 DMA 请求线上的变换都最终导致执行一个 DMA 周期。恰好相反，在 DMA 请求被处理识别的总线周期结束之前的某一时刻，DMA 请求必须是活跃的。若在该时刻之前 DMA 请求信号变成不活跃，则没有 DMA 周期会被执行。

§14.4.5 DMA 响 应

80186 不产生直接的 DMA 响应信号，它是用直接对请求 DMA 的设备执行一个读或写来响应的。在需要的情况下，DMA 响应信号可以通过对一个地址的译码来产生，或是仅用一个 $\overline{PCS}$ 信号来响应（见图 14.36）。注意，ALE 必须被用来作为 DACK 的一个因素，因为当片选信号变活跃时，并不能保证地址已经稳定。需要这样做，是因为当 $\overline{PCS}$ 变活跃时，若地址尚未稳定，则当地址线改变状态时，在DACK 发生电路的输出端就会出现假信号。一旦 ALE 已变成低电平，地址就保证已经稳定了至少 t_{AVAL}(30ns)。

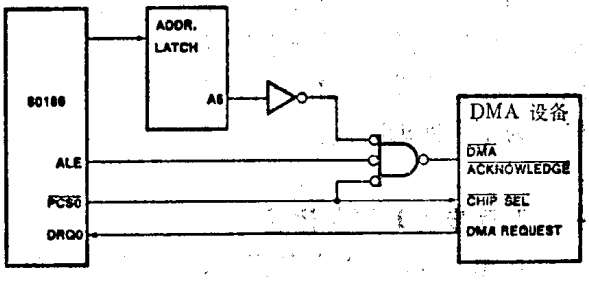


图 14.36 从 80186 中合成出 DMA 响应信号

§14.4.6 内部产生的 DMA 请求

80186 中有两种内部产生的 DMA 请求，即由一个集成在 80186 内部的部件启动的传送。这两种类型的传送，是由计时器 2 产生的 DMA 请求，或是由 DMA 通道自己产生的 DMA 请求启动的。

DMA 通道可以这样编程：即一旦计时器 2 达到它的最大计数，就能产生一个 DMA 请求。这个特性是通过设置在 DMA 通道控制寄存器中的 TDRQ 位来选定的。以这种方式产生的 DMA 请求，将被锁存在 DMA 控制器中，以确保计时器请求产生以后不被清除。这种请求只有运行该 DMA 周期或用清除在两个 DMA 控制寄存器的 TDRQ 位的方法才能被清除。在这种方式的任何 DMA 请求产生以前，定时器 2 必须是被初始化的并且是被启动的。

由任一通道运行的计时器请求 DMA 周期，将复位该计时器请求。这样，若两个通道都使用它来请求一个 DMA 周期，则对计时器 2 的每次超时，将只执行一个 DMA 通道传送。在 DMA 控制器中锁存一位计时器 DMA 请求的另一个意义是，若在一个 DMA 通道有机会运行一个 DMA 传送之前，又发生了另一个计时器 2 超时，则第一个请求将被丢失，即尽管计时器超时了二次，但只会发生一次 DMA 传送。

DMA 通道，也能被编程为为它自己提供 DMA 请求。在这种方式中，DMA 传送周期将

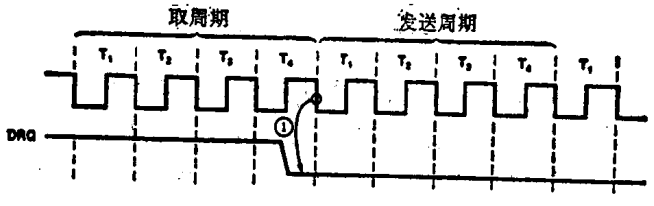
以最大的总线带宽连续地运行，直到完成被编程的 DMA 传送次数为止（传送次数放在 DMA 传送寄存器中）。这种方式，是通过对在 DMA 控制寄存器中用于非同步的传送的同步位，进行编程而选定的。在这种方式下，DMA 控制器将垄断 CPU 总线，即在进行 DMA 传送时，CPU 将不能执行取操作码、存贮器操作等。同样，DMA 将不管在 DMA 控制寄存器中的 TC 位是什么状态，一直把在计数寄存器中指出的最大传送次数执行完。

§14.4.7 外部同步的 DMA 传送

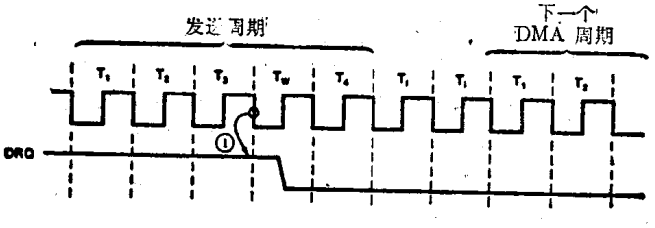
80186 有两种外部同步的 DMA 传送，它们是由外部设备、而不是由集成的计时器 2 提出的 DMA 传送请求，或由 DMA 通道本身提出的 DMA 请求（用非同步方法传送），亦即源同步的和目同步的传送。这些方式是通过对在 DMA 通道控制寄存器中的同步位，进行编程而选定的。这两种传送的唯一区别，是对 DMA 请求引脚进行检查，以决定在当前正在执行的 DMA 传送之后，是否要立即再执行一个 DMA 传送的时机。在源同步传送方式中，两个源同步的 DMA 传送，可以一个紧接着一个进行，但在目同步的传送方式中，则将有一定数量的空闲周期将被自动地插在两个 DMA 传送之间，以便为 DMA 请求设备提供把它的 DMA 请求驱动为不活跃的时间。

一、源同步的 DMA 传送

在源同步的 DMA 传送中，DMA 数据的源请求 DMA 传送。例如从磁盘中把数据读入主存贮器。在这种类型的传送中，请求传送的设备，在 DMA 传送的取周期期间就被读出。由于从取样 DMA 请求到 DMA 传送正式开始要 4 个 CPU 时钟周期，并且一个总线周期，至少需要 4 个时钟周期，所以在 DMA 请求引脚上对另一个 DMA 传送请求进行取样的最早时机，将是一个 DMA 传送的发送周期开始的时候。这样，在 DMA 请求设备发出 DMA 请求到它收到对它的请求的响应（在 DMA 取周期的 T_2 开始时）之间，有超过 3 个 CPU 时钟周期的时间，在这段时间里，若不需要执行另一个 DMA 传送，则它就必须把它的请求信号驱动为不活跃（见图 14.37）。



80186 决定：　1. 当前的 DMA 源同步的传送将不立即跟着另一个 DMA 传送



80186 决定：　1. 当前的 DMA 目同步的传送将立即跟着另一个 DMA 传送

图 14.37　源和目同步的 DMA 请求定时

二、目同步的 DMA 传送

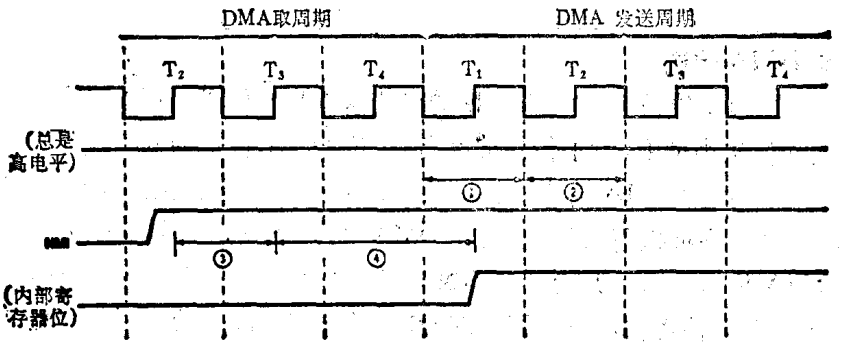
在目同步的 DMA 传送中，DMA 数据的目请求 DMA 传送，例如，从主存贮器中把数据写入磁盘。在这种传送类型中，请求传送的设备，在该 DMA 传送的发送周期期间被写入。这样就产生了一个问题，因为 DMA 请求设备，要在该 DMA 传送结束之前的 3 个时钟周期，才会收到该 DMA 周期正在被执行的消息(设在 DMA 传送的发送周期没有插入等待状态)，而决定在当前的 DMA 传送之后，是否要立即执行另一个 DMA 传送却要 4 个时钟周期。为了解决这个问题，DMA 部件将在每个目同步的 DMA 传送之后，放弃 CPU 总线至少 2 个 CPU 时钟周期，以便使 DMA 请求设备，在不要立即进行另一个 DMA 传送时，有足够的时间去降下 DMA 请求信号。当总线被 DMA 部件放弃时，CPU 可以重新进行总线操作(例如取指令、存贮器或 I/O 读或写等)。这样，一般来说，在目同步的 DMA 传送中，将会插入由 CPU 启动的总线周期。若不存在 CPU 的总线操作，则 DMA 部件在一个 DMA 传送的发送周期和下一个 DMA 传送的取周期之间，只插入 2 个 CPU 时钟周期。这表示，不管在总线周期中插入了多少个等待状态，在发送周期结束之前，DMA 目请求设备必须把它的 DMA 请求信号降下至少 2 个时钟周期。图 14.37 表示了 DMA 请求降下得太晚了因而不能避免立即产生另一个 DMA 传送。任何插入 DMA 传送的发送周期的等待状态，将会使从发送周期开始到为另一个 DMA 传送请求进行取样的时间延长。这样，若一个设备在收到 80186 的 DMA 响应信号以后，用来降下它的 DMA 请求信号所需的时间大于具有 0 个等待状态的 80186 的最大时间 (100ns)，就要把等待状态插入 DMA 周期，以延长在收到 DMA 响应信号以后，DMA 请求设备所能用来降下它的 DMA 请求信号的时间。表 14.4 表示了在 T_2 开始和取样到 DMA 请求的时刻之间，在 DMA 发送周期中插入等待状态时所需的时间。

表 14.4 DMA 请求信号不活跃定时

等待状态数	从 T_2 开始到 DRQ 不活跃的最大时间 (ns)
0	100
1	225
2	350
3	475

§14.4.8 DMA 暂停和 NMI

一旦 80186 接收到一个不可屏蔽中断，所有的 DMA 活动都将在当前的 DMA 传送结束之后挂起。这是由 NMI 自动地设置在中断控制器状态寄存器中的 DMA 暂停(DHLT)位而实现的。防止出现一个 DMA 周期所需要的 NMI 定时表示在图 14.38 中。在 NMI 服务结束之后，DHLT 位应当由程序员来清除，在清除以后，DMA 活动就在它原来停止的地方重新开始，即在这个过程中 DMA 寄存器一个也没有被修改过。DMA 的暂停位，在 NMI 服务之后不是自动复位的，但在执行 IRET 指令时却是自动复位的。DMA 暂停位也可以由程序员设置，它可以用来在执行关键的代码段时防止出现 DMA 操作。



1. DMA 请求同步
2. 决定：要运行 DMA 周期吗？　答案：不要，因为虽然 DMA 请求信号是活跃的，但 DHLT 置位(来自 NMI 请求)
3. NMI 同步时间
4. 从同步的 NMI 到 DHLT 置位的逻辑延迟时间(注：DHLT 是在中断控制状态寄存器中)

图 14.38　NMI 和 DMA 接口

§14.4.9　DMA 接口举例

一、8272 软磁盘接口

图 14.39 是一个 80186 和 8272 软磁盘控制器的 DMA 接口。这个图表示了一个典型的 DMA 设备怎样才能与 80186 接口，接口的软磁盘软件驱动程序在 §14.11.3 中给出。

8272 的数据线，是通过缓冲器连接到 80186 的 AD_0—AD_7 引脚的。需要这些缓冲器，是因为 8272 不能尽快地浮空它的输出驱动器，以避免与 80186 在从 8272 中读出以后所驱动的

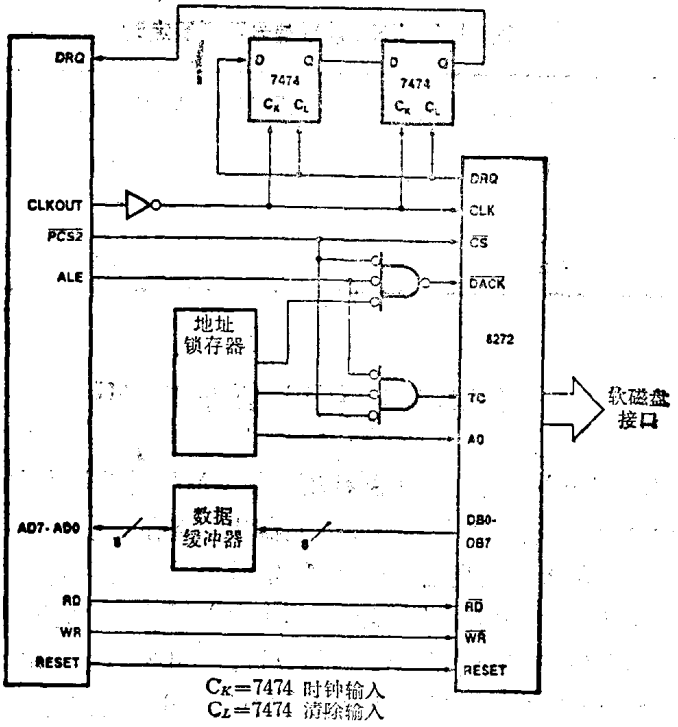


图 14.39　8272/80186 DMA 接口举例

地址信息冲突。

对 8272 的 DMA 响应，是由一个处在分配给 $\overline{PCS2}$ 的区域中的一个地址译码器驱动的。若 $\overline{PCS2}$ 被分配为在 I/O 单元 0500H 和 057FH 之间活跃，则一个对 I/O 单元 0500H 的访问，将只启动片选信号，而对 I/O 单元 0510H 的访问，则将启动片选信号和 DMA 响应。ALE 必须作为 DACK 产生逻辑的一个因素，因为当片选信号变活跃时，并不能保证地址已经稳定。若不使用 ALE，当地址输出改变了状态，而片选信号仍是活跃时，DACK 产生电路就会进行误操作。

8272 的 TC 信号，是由一个与产生 DACK 信号的电路十分相似的电路驱动的(除了保留输出以外)。这条线上的信号，用来在一条 8272 命令还没有执行完时终止该命令。因此，在这种情况下，给 8272 的 TC 输入，是由软件驱动的。另一种驱动 TC 的方法，是把 DACK 信号连接到 80186 的某一个计时器，并把该计时器编程为在执行完一定数量的 DMA 周期以后，就向 8272 输出一个脉冲。

上面的讨论，是在用一根 80186 的 $\overline{PCS}$ 线产生所有的 8272 选择信号的假设下进行的，若有多个片选信号可供使用，则 80186 产生的不同的 $\overline{PCS}$ 信号，就能为不同的功能所使用。例如 $\overline{PCS2}$ 可以用来选择该 8272，$\overline{PCS3}$ 可以用来驱动 8272 的 DACK 信号等。

从 8272 到 80186 的 DMA 请求，被两个时钟片段所延迟。这是 8272 的 t_{RQR} (从 DMA 请求到 DMA $\overline{RD}$ 变活跃的时间)所需要的，它的最小值是 800ns。这需要 6.4 个 80186 CPU 时钟周期(8MHz)大大超过了 80186 提供的最小值 5 个时钟周期(从 DMA 总线周期开始是 4 个总线周期，从 $\overline{RD}$ 将要变活跃的 DMA 周期的 T_2 开始，是 5 个时钟周期)。两个触发器给这个响应时间，加了两个完整的 CPU 时钟周期。

在 DACK 被送到 8272 200ns 以后，DMA 请求信号就将消失。在一个 DMA 写周期期间(即一个目同步的传送)，若在发送周期不插入等待状态，则这个时间对于防止立即产生另一个 DMA 传送来说是太长了，也就是说，在这样一段时间里，有可能会产生一个不希望出现的 DMA 传送。因此，无论 8272 的数据访问参数是什么，这个接口至少要插入一个等待周期。

二、8274 串行通信接口

图 14.40 是一个 8274 同步/异步串行片子/80186 DMA 接口的例子。该 8274 接口比 8272 接口简单，因为它不需要产生 DMA 响应信号，并且 8274 在一个 DMA 请求和该 DMA 读或写周期之间所需要的时间，不象 8272 那么长。80186 在 DMA 方式下，使用 8274 的串行驱动程序在 §14.11.3 中给出。

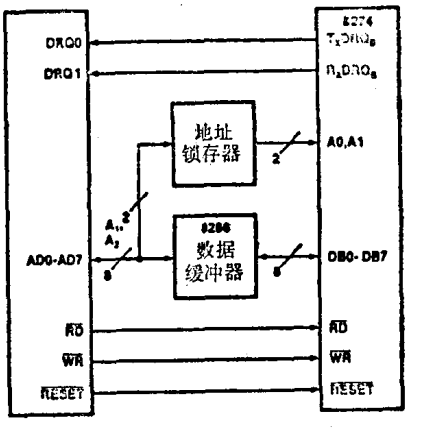


图 14.40　8274/80186 DMA 接口的例子

8274 的数据线，通过缓冲器，再连接到 80186 的 AD_0—AD_7 引脚。同 8272 的情况一样，需要这些缓冲器并不是总线驱动能力的问题，而是 80186 在地址/数据总线上驱动地址信息以前，8274 不能浮空它的驱动器。若 8274 和 8282 都包含在同一个 80186 系统中，它们就能共享同一个数据缓冲器(和其他在系统中的外围设备一样)。

8274 不需要 DMA 响应信号。在 8274 产生 DMA 请求信号之后，第一个从 8274 数据寄存器中读出或对该寄存器写入的操作，就能清除该 DMA

请求。在一个 DMA 写期间，从控制信号 ($\overline{RD}$ 或 $\overline{WR}$) 变活跃到 8274 降下它的 DMA 请求信号之间的时间是 150ns。为了使这个接口的操作能正常地进行，对这种 DMA 写周期至少要插入一个等待状态。

§14.5 计时器部件接口

80186 包含了一个计时器部件，该部件提供了 3 个独立的 16 位计时器。这些计时器的操作是独立于 CPU 的。其中两个具有输入和输出引脚，以便能计数外部事件和产生任意的波形，第三个计时器用作计时器，其他两个计时器的前置计时器或作为一种 DMA 请求的源。

§14.5.1 计时器操作

80186 中的内部计时器部件，由一个计数器元件来组织，时间被分路送到三个寄存器体中

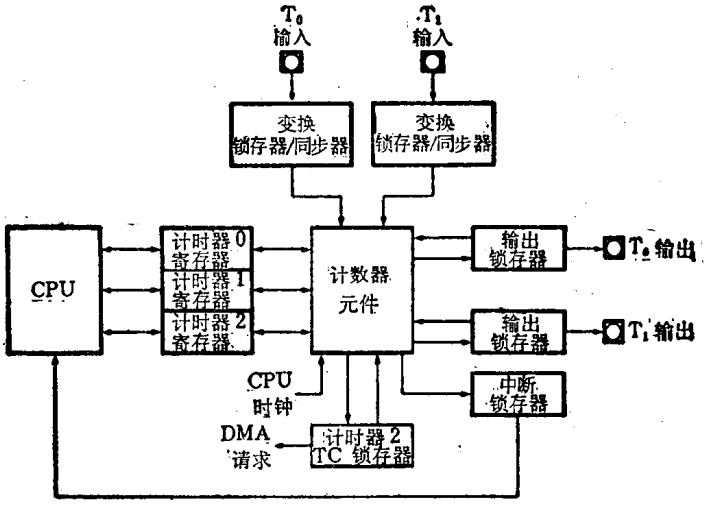
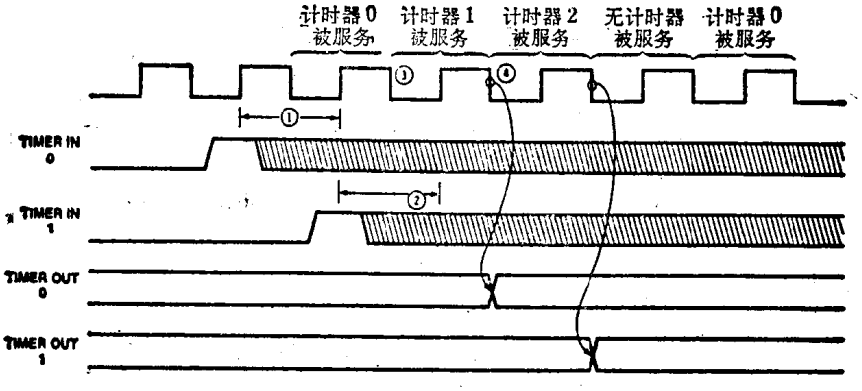


图 14.41 80186 计时器模块



1. 计时器处于 0 消除时间　2. 计时器处于 1 消除时间　3. 修改过的计数值写入 80186 计时器 0 计数寄存器
4. 修改过的计数值写入 80186 计时器 1 计数寄存器

图 14.42 80186 计数器元件分路和计时器输入同步

去,其中每一个体都含有不同的控制和计数值。这些寄存器体在计数器元件和 80186 CPU 之间是双向端口的(见图 14.41)。图 14.42 是计数器元件的序列和在输入、输出信号上的约束。若 CPU 修改计时器寄存器中的一个,这个变化就会在下次该寄存器值被送到计数器元件时,影响计数器元件。在经过计时器寄存器体的计数器元件序列和经过若干个 T 状态的总线接口部件序列之间,不存在联系。计时器操作和总线接口操作是完全异步的。

§14.5.2 计时器寄存器

每一个计时器,由一个寄存器块控制(见图 14.43)。不论该计时器是否正在操作,这个寄存器块中的每一个寄存器,都能被读出和写入。所有的处理器对这些寄存器的访问,要同步于所有计数器元件对这些寄存器的访问,这表示不可能读出一个只修改了其中一半位的计数寄存器的值。由于这种同步的存在,任何对计时器寄存器的访问,都将被自动地插入一个等待周期。同 DMA 部件不一样,封锁对计时器寄存器的访问,并不能阻止计时器的计数器元件访问计时器寄存器。

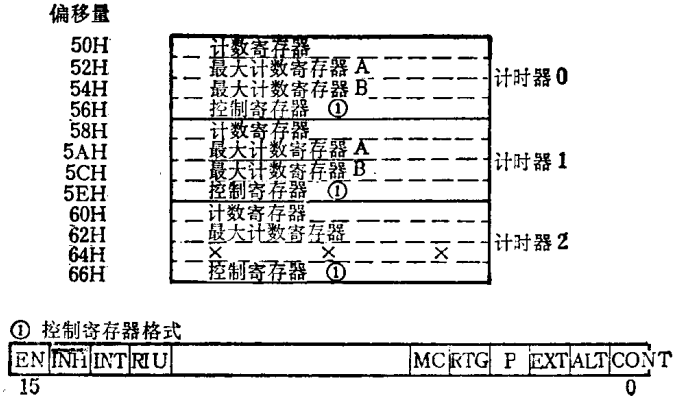


图 14.43 80186 计时器寄存器布局

每一个计时器,具有一个 16 位的计数寄存器。该寄存器在每个计时器事件之后递增。一个计时器事件可以是在外部引脚上一个低电平到高电平的交换(对计时器 0 和 1 而言),一个 CPU 时钟变换(由于计数器元件分路,所以要除以 4)或一个计时器 2 的时间超出(对计时器 0 和 1 而言)。由于计数器寄存器是 16 位宽的,所以一个计时器/计数器就可以记多达 65536 (2^{16})个计时器事件。不管相应的计时器是否正在操作,该寄存器都可以被读出或被写入。

每个计时器,含有一个最大计数寄存器。一旦计时器的计数寄存器值与最大计数寄存器值相等,该计数寄存器值将被复位为 0,即最大计数值将不再存放在计数寄存器中。这种最大计数值可在该计时器正在操作时被写入。最大计数值 0 隐含着一个值为 65536 的最大计数,最大计数值 1 隐含着值为 1 的最大计数。应当知道,在计数值和最大计数寄存器值之间,只进行相等检验,即若在计数寄存器中的值大于在最大计数寄存器中的值的情况下,该计数值将不会被清除。当然,这种情况只有在程序员的干涉下,才可能出现。例如把在计数寄存器中的值,设置得比在最大计数寄存器中的值还要大,或把在最大计数寄存器中的值设置得比计数寄存器中的值还要小时,就会出现这种情况。如果出现了上述情况,该计时器将计数到最大的可能计数值 (FFFFH),并递增到 0,然后再计数到在最大计数寄存器中所规定的值。计时器控

制寄存器中的 TC 位，在计数器溢出到 0 时不会被置位，同时，也不会从计时器部件产生中断。

计时器 0 和 1，每个都含有一个附加的最大计数寄存器。当两个最大计数寄存器都使用时，计时器将先计数到在最大计数寄存器 A 中的值，复位到 0；再计数到在最大计数寄存器 B 中的值，然后再复位到 0。在计时器控制寄存器中的 ALT 位，决定使用一个最大计数寄存器还是两个都使用。若这一位是低电平，则只使用最大计数寄存器 A，最大计数寄存器 B 被忽略。若该位是高电平，则最大计数寄存器 A 和 B 都使用。计时器控制寄存器中的 RIU (Register In Use) 位，指示当前正在使用那一个最大计数寄存器。最大计数寄存器 A 正被使用时，这一位是 0，最大计数寄存器 B 正被使用时，这一位就是 1。这个 RIU 位是只读的，它不受任何对计时器控制寄存器执行的写入操作的影响，并且在单个最大计数寄存器方式下，总是被读出为 0 (因为只有最大计数寄存器 A 将被使用)。

一旦计时器的计数值到达了最大计数值，每个计时器就能产生一个中断。这就是说，只要达到了在最大计数寄存器 A 或 B 中的值，就会产生一个中断。另外，只要计时器计数达到了最大计数值，计时器控制寄存器中的 MC(Maximum Count) 位就被置位。这个位是不会被自动地清除的，即清除这一位需要程序员干预。若一个计时器在第一个中断请求被服务之前，又提出了第二个中断请求，则第一个向 CPU 提出的中断请求将丢失。

在每个计时器的控制寄存器中，有一个 EN(ENable) 位，这一位用来启动计时器进行计数。计时器只有在这一位置位的情况下，才计数计时器事件。在这一位被复位时，所出现的任何计时器事件都被忽略。任何对计时器控制寄存器进行的写操作，仅在 INH(INHibit) 位也置位的情况下，才能修改 EN 位。换一句话说，要使对计时器控制寄存器的写操作不能修改 EN 位，只要 INH 位不置位就可以了。计时器控制寄存器中的 INH 位，起着使计时器 EN 位能被有选择地进行更新的作用。在一个向计时器控制寄存器进行的写操作中，INH 位的值并不被存放，它将总是被读出为 1。

每一个计时器的计时器控制寄存器中，有一个 CONT(CONTinuous) 位。若这一位被清除，则该计时器 EN 位在每个定时周期的末尾，将被自动地清除。若使用一个最大计数寄存器，定时周期的末尾，将出现在计数值达到了在最大计数寄存器 A 中的值之后，该计数值被复位为 0 的时候。若使用两个最大计数寄存器，定时周期的末尾，将出现在计数值达到了在最大计数寄存器 B 中的计数值之后，该计数值被复位为 0 的时候。若 CONT 位被置位，则在计时器控制寄存器中的 EN 位，再也不能自动地复位。这样，在每一个定时周期之后，就将自动地开始另一个定时周期。例如，在单个最大计数寄存器方式下，计时器将计数到在最大计数寄存器 A 中的值，复位为 0，再计数到在最大计数寄存器 A 中的值，再复位为 0，如此循环，直到无穷。在双最大计数寄存器方式下，计时器将计数到在最大又数寄存器 A 中的值，复位为 0，计数到在最大计数寄存器 B 中的值，复位为 0；再计数到在最大计数寄存器 A 中的值，再复位为 0，如此循环直到无穷。

§14.5.3 计时器事件

每个计数器，都计数计数器事件。所有的计数器，都可以把一个 CPU 时钟的变换，作为一个事件。由于计数器元件是分路的，所以计数器的值，每 4 个 CPU 时钟递增一次。对计时器 2 来说，这是它可以使用的唯一计数器事件。对计数器 0 和 1，这种事件由清除在计时器控制

寄存器中的 EXT(EXTernal) 和 P(Prescaler) 位来选择.

计时器 0 和 1,把计时器 2 到达它的最大计数,作为一个计时器事件. 这可以通过清除在该计时器控制寄存器中的 EXT 位和设置在该寄存器中的 P 位来选择. 在这样操作的时候,每当计时器 2 到达它自己的最大计数而复位到 0 的时候,该计时器就将递增. 要注意,计时器 2 必须被初始化,并为其他计时器的要递增的值而运行.

计时器 0 和 1, 也能被编程为计数在外部输入引脚上的低电平到高电平的变换. 外部引脚上的每个变换,在它被送到计时器电路以前,都被同步于 80186 时钟,所以这种变换是异步的. 计时器计数的上述变换,即输入值必须先变成低电平然后再变成高电平,才能引起计时器的递增. 在这个输入引脚上的任何变换, 都将被锁存. 若一个变换在相应的计时器不是正在被计数器元件服务时出现,则在输入引脚上的该变换将被记住, 以便在计时器取得服务时, 把该输入变换计入. 由于计数器元件是分路的, 所以计时器能够计数的最大速率,是 CPU 时钟速率的 1/4(对 8 MHz 的 CPU 时钟是 2 MHz).

§14.5.4 计时器输入引脚操作

计时器 0 和 1,各有一个计时器输入引脚,引脚上的低电平到高电平的变换,都被同步、锁存,并在特定的计时器正被计数器元件服务时,送到计数器元件.

该输入上的信号,能够以 3 种不同的方式,影响计时器的操作. 使用引脚信号所进行的操作,是由在计时器控制寄存器中的 EXT 和 RTG(ReTriGger)位决定的. 若 EXT 位置位, 在计时器被启动的情况下 (EN 位置位), 输入引脚上的变换,就会引起计时器的值递增. 这样,计时器就计数了外部事件;若 EXT 位被清除,则所有计时器的递增,都将是由 CPU 时钟、或由计时器 2 的定时输出所引起的. 在这种方式下, RTG 位决定输入引脚上的信号是否将启动计时器操作,或它是否会重新触发计时器操作.

若 EXT 位是低电平, 并且 RTG 位也是低电平,则在计时器输入引脚是高电平、并且在计时器控制寄存器中的 EN 位置位时,该计时器将只计数内部的计时器事件. 注意, 在这种方式中,该引脚是电平敏感的而不是边沿敏感的. 启动计时器的操作,并不需要在计时器输入引脚上的低电平到高电平的变换. 若该输入和高电平相接, 计时器就将持续地被启动. 计时器的启动输入信号,是完全独立于在计时器控制寄存器中的 EN 位的. 为了使计时器能计数,这两个信号都必须是高电平. 实时时钟和波特率发生器,就是在这种方式下使用计时器的例子.

若 EXT 位是低电平,并且 RTG 位是高电平,即计时器的动作就象数字单发. 在这种方式下,计时器输入引脚上的每个从低电平到高电平的变换,都将导致计时器复位为 0. 若计时器是启动的 (EN = 1), 计时器操作就将开始 (该计时器将计数 CPU 时钟变换, 或计时器 2 的超时). 在一个记时周期结束的时候,计时器操作将停止,这就是说,在达到最大计数寄存器 A 中的值,计时器计数值被复位为 0 时; 或在达到最大计数寄存器 B 中的值, 计时器计数值被复位为 0 时 (分别对应于使用一个或两个最大计数寄存器的情况), 计时器的操作就将停止. 若在计时器周期结束之前,输入引脚上又出现了一个从低电平到高电平的变换,则该计时器将复位为 0,并且重新开始计时周期. 在这种情况下,计时器控制寄存器中 CONT 位的状态是不起作用的. 计时器控制寄存器中的 RIU 位, 将不会被这种输入的变换所改变. 若 CONT 被清除, 则计时器的 EN 位, 在计时器周期的末尾将被自动地清除. 这表示在输入引脚上任何附加的

变换，都将被计时器忽略。若 CONT 置位，对在输入引脚上的每一个低电平到高电平的变换，不论计时器是否已经到达了一个计时器周期的末尾，计时器都将复位为 0，并且开始另一个计时周期。这是因为该计时器的 EN 位，在计时周期的末尾，不会被清除的缘故。警告时钟超时信号或中断，是计时器在这种方式下使用的一个例子。

§14.5.5 计时器输出引脚操作

计时器 0 和计时器 1，各含有一个计时器输出引脚。这个引脚在程序员的选择下，可以实现两种功能：第一种功能是用一个单脉冲指示一个计时周期的结束；第二种功能是用一个电平指示当前正在使用的最大计数寄存器。

不论是使用计时器的内部还是外部定时，计时器的输出操作都可以描述如下：若使用的是外部定时，则在该计时器输入脚上的变换、和该变换在这个计时器输出脚上得到反应之间的时间，依赖于与正由计数器元件进行服务的计时器相关的输入引脚上该变换出现的时间。

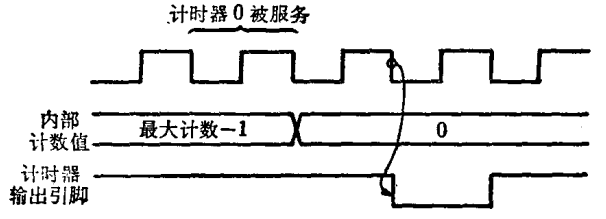


图 14.44 80186 计时器输出信号

当计时器以单个最大计数器方式操作时 (ALT = 0)，在已到达最大计数值、并且该计时器已被计数器元件服务过以后的一个时钟周期中，该计时器的输出引脚将会变成低电平(见图 14.44)。在把计时器用作波特率发生器的时候，这种方式是很有用的。

在计时器被编程为用双最大计数寄存器方式时 (ALT = 1)，计时器输出引脚，指示那一个最大计数寄存器正在被使用。若最大计数寄存器 B 正在被使用，则输出引脚为低电平，若最大记数寄存器 A 正在被使用，则输出引脚为高电平。若计时器被编程为连续工作方式 (CONT = 1)，在输出引脚上就能产生出任意占空度的波形。例如，若最大计数寄存器 A 中含有 10，而最大计数寄存器中含有 20，就能产生一种占空率为 33% 的波形。

§14.5.6 80186 计时器应用实例

在大多数要使用分立的计时器电路的应用中，都可以使用 80186 的计时器。这些应用可以是实时时钟、波特率发生器或事件计数器等。

一、80186 计时器实时时钟

在 §14.11.4 中所示的程序实例，表示 80186 的计时器正被用来同 80186 的 CPU 一起，构成一个实时时钟，如图 14.45 所示。在这种结构中，计时器 2，被编程为每毫秒中断 CPU 一次，然后由 CPU 递增在存贮器中的时钟变量。

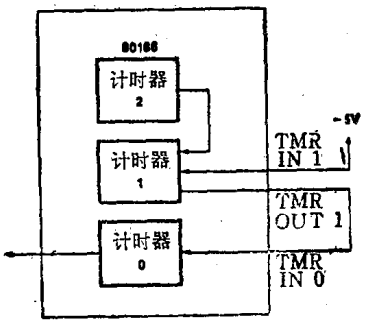


图 14.45 80186 实时时钟

二、80186 计时器波特率发生器

80186 计时器，也能用作串行通讯控制器 (例如 8274) 的波特率发生器。图 14.46 表示了

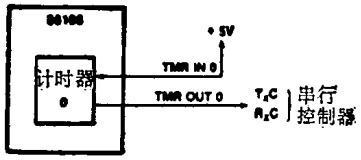


图 14.46　80186 波特率发生器

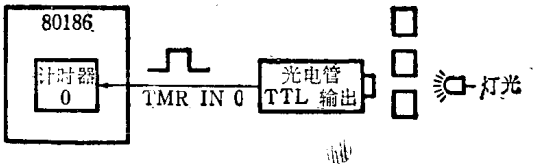


图 14.47　80186 计时器用作灯光闪烁次数的计数器

这种波特率发生器的构成，使计时器按波特率发生器要求操作的代码在 §14.11.4 中。

三、80186 计时器事件计数器

80186 计时器，还可以用来计数事件。图 14.47 表示了把 80186 计时器用作为计数灯光闪烁次数的一个实例。在该实例中，灯光的闪烁次数，可以直接从计时器的计数寄存器中读得，因为灯光每闪烁一次，就会引起计时器的计数值递增 1，使 80186 计时器在这种方式下操作的代码在 §14.11.4 中。

§14.6　80186 中断控制器接口

80186 含有一个集成的中断控制器。在一般系统中，这个部件执行中断控制器的任务。这些任务包括对中断请求的同步、判优以及提供所请求中断的类型向量，以作为对 CPU 中断响应的回答。它可以成为两个外部 8259 A 中断控制器的主控制器，也能够成为一个外部中断控制器的从控制器，以便能同 iRMX 86 操作系统和 80130/80150 操作系统固件相兼容。

§14.6.1　中断控制器模块

图 14.48 是集成的中断控制器的方框图，它含有寄存器和控制元件。这个控制器有 4 个输入作为与外部的接口，其功能根据程序指定的中断控制器的工作方式而改变。和 80186 其他的外围寄存器一样，中断控制寄存器中的寄存器，也能在任何时候为 CPU 所读写。

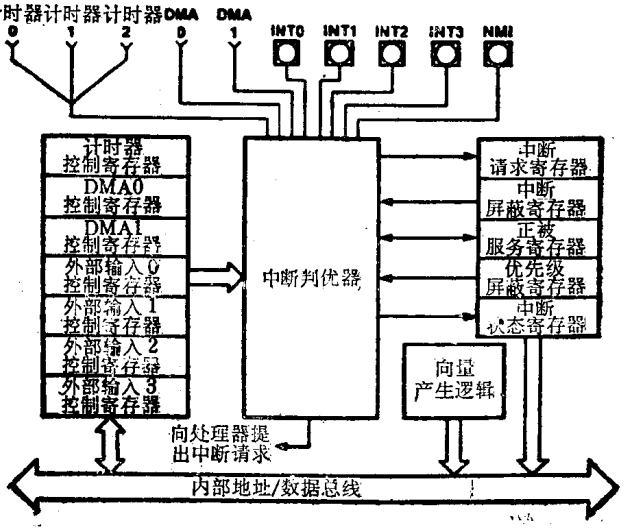


图 14.48　80186 中断控制器方框图

§14.6.2 中断控制器操作

80186 集成的中断控制器，以两种主要的方式进行操作。这两种方式是非 iRMX 86 方式(以后就称为主控制器方式)和 iRMX 86 方式。在主控制器方式(非 iRMX 86 方式)中，集成的中断控制器，起着系统的主中断控制器作用，而在 iRMX 86 方式中，该控制器起着一个作为系统的主中断控制器的外部中断控制器的从控制器的作用。在这两种方式之间，一些中断控制器寄存器和中断控制器的引脚的定义有些不同，但该中断控制器的基本特性和功能仍保持不变。两种方式之间的区别是，在主控制器方式下，该中断控制器把它的中断输入，直接送到 80186 的 CPU，而在 iRMX 86 方式下，该中断控制器把它的中断输入送到一个外部控制器(然后由该外部中断控制器把它的中断输入送给 80186 CPU)。通过设置外围控制块指示器中的 iRMX 方式位，就能使中断控制器在 iRMX 86 方式下操作(参见 §14.11.1)。

§14.6.3 中断控制器寄存器

中断控制器，具有许多用来控制它的操作的寄存器 (见图 14.49)。其中的一些，在中断控制器的两种主要工作方式之间，改变它们的功能。在下面的叙述中，将会指出在两种方式下寄存器功能的区别。若不特别指出，就说明那种功能在两种主要方式下都是相同的。各种中断控制器寄存器之间相互作用的方法，在图 14.57 和图 14.58 的流程图中表示。

主控制器方式	偏移地址	iRMX86 方式
INT3 控制寄存器	3EH	①
INT2 控制寄存器	3CH	①
INT1 控制寄存器	3AH	计时器 2 控制寄存器
INT0 控制寄存器	38H	计时器 1 控制寄存器
DMA1 控制寄存器	36H	DMA1 控制寄存器
DMA0 控制寄存器	34H	DMA0 控制寄存器
计时器控制寄存器	32H	计时器 0 控制寄存器
中断控制器状态寄存器	30H	中断控制器状态寄存器
中断请求寄存器	2EH	中断请求寄存器
正被服务寄存器	2CH	正被服务寄存器
优先级屏蔽寄存器	2AH	优先级屏蔽寄存器
屏蔽寄存器	28H	屏蔽寄存器
查询状态寄存器	26H	①
查询寄存器	24H	①
EOI 寄存器	22H	专用 EOI 寄存器
①	20H	中断向量寄存器

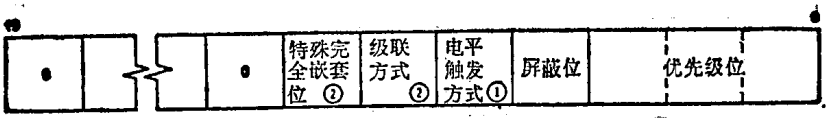
1. 在这种方式中不提供：写入的值可能被存放，也可能不被存放

图 14.49 80186 中断控制器寄存器

一、控制寄存器

向 80186 提出中断请求的每一个中断源，在内部中断控制器中，都有一个控制寄存器。这些寄存器都含有 3 个决定优先级的位，用来为提出中断请求的设备，选择 8 个优先级中的一种(0 表示最高优先级，7 表示最低级)，还有 1 个中断屏蔽位，该位用来允许中断(见图 50)。当中断屏蔽位是低电平时，该中断被允许，当中断屏蔽位是高电平时，该中断被屏蔽。

在 80186 集成的中断控制器中，有 7 个控制寄存器。在主控制器方式下，其中的 4 个，作为



① 这一位只出现在 INT0—INT3 控制寄存器中

② 这些位只出现在 INT0—INT1 控制寄存器中

图 14.50 中断控制器控制寄存器

外部中断输入，对两个 DMA 通道各使用了 1 个寄存器，还有 1 个用来接收计时器的中断。在 iRMX 86 方式下，外部中断输入不使用，从而使每个计时器都有它自己单独的控制寄存器。

二、请求寄存器

中断控制器，含有一个中断请求寄存器（见图 14.51）。该寄存器含有 7 个活跃位，其中每一位对应一个中断控制寄存器。一旦与某个特定的中断控制寄存器相连的中断源提出一个中断请求，在中断请求寄存器中的相应位就置位。这种操作是不受是否允许中断，或是否有足够的优先级来引起处理器中断等条件的约束的。在这个寄存器中与集成的外围器件（DMA 和计时器部件）相连的位，是可被读出或被写入的，而那些与外部中断引脚相连的位，则只能被读出（对它们写入的位，不会被存贮起来）。在中断被响应以后，这些中断请求位就被自动地清除。

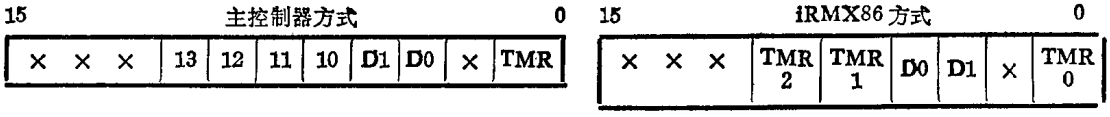


图 14.51 80186 中断控制器的中断请求和屏蔽寄存器的格式

三、屏蔽寄存器和优先级屏蔽寄存器

中断控制器，含有一个屏蔽寄存器（见图 14.51）。该寄存器为每一个与中断控制寄存器相联系的中断源，提供了一个屏蔽位。在该屏蔽寄存器中某一个与中断源所对应的位，就是对应于该中断源的中断控制器寄存器中的屏蔽位，也就是说，修改在控制寄存器中的屏蔽位，就会使屏蔽寄存器中相应于该控制寄存器的位也被修改。

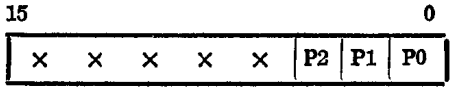


图 14.52 80186 中断控制器优先级屏蔽寄存器格式

中断控制器，还含有一个优先级屏蔽寄存器（见图 14.52）。这个寄存器含有 3 位，指示着当前正在被服务的中断的优先级。当一个中断被响应时（处理器运行中断响应，或是处理器正在读中断查询寄存器时），这些位就会被设置为请求中断的设备的优先级，但不能低于先前编程入这些位的优先级。这样就能防止较低优先级的中断（由中断控制寄存器中的优先级位所表示）来中断处理器。当 CPU 向中断处理器发出中断结束命令时，这些位就自动地设置为下一个最低的中断优先级（或在没有中断在等待处理时，设置为全 1，表示允许任何优先级的中断）。这个寄存器可以被读出或写入。

四、正被服务寄存器

中断控制器含有一个正被服务寄存器（见图 14.51）。正被服务寄存器中的每一位，对应着一个中断控制寄存器。当与某个控制寄存器相连的设备被处理器响应时，在正被服务寄存器中与该控制寄存器对应的位就被置位。当 CPU 向中断控制器发出结束中断命令时，该位就被复位。这个寄存器可以被读出也可以被写入，即 CPU 能在不发生中断的情况下，设置正被服务位，也可以不使用中断控制器的 EOI 功能，就把它们复位。

五、查询和查询状态寄存器

中断控制器，含有一个查询寄存器和一个查询状态寄存器（见图 14.53）。这两个寄存器含有相同的信息，它们都有一位指示是否有一个中断正在等待处理。若收到了一个满足优先级要求的中断，这一位就置位。当该中断被响应时，这一位就自动地清除。在有中断等待处理时，它们含有具有最高优先级的等待中断的信息。

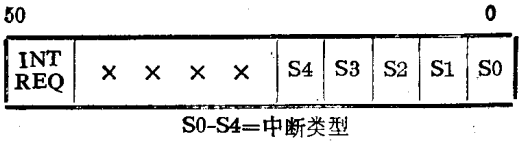


图 14.53　80186 查询和查询状态寄存器格式

读查询寄存器，中断控制器就响应了等待的中断，就象通过中断响应周期处理器响应中断一样，处理器实际上并不执行任何中断响应周期，也不通过中断向量表作向量转移。仅有的操作就是对中断控制器中的中断请求、正被服务和优先级屏蔽寄存器作适当的设置。读查询状态寄存器将仅仅传送查询位的状态，而不修改任何中断控制器寄存器。这些寄存器是只读的，即写向它们的数据将不被存放起来。这些寄存器在 iRMX 86 方式中是不提供的，因而这些寄存器中的状态位也是没有定义的，但是在 iRMX 86 方式下，访问查询寄存器单元会使中断控制器去"响应"中断（即将会置位正被服务位和优先级屏蔽寄存器位）。

六、中断结束寄存器

中断控制寄存器，含有一个中断结束寄存器（见图 14.54）。程序员通过写这个寄存器，把中断结束命令发给中断控制器。在收到中断结束命令以后，中断控制器就自动地把这个中断的正被服务位和优先级屏蔽寄存器的位复位。写入这个寄存器的字值，决定中断结束命令是特别的，还是非特别的。非特别的中断结束命令，是由写入中断结束寄存器的字中的非特别位置位来指定的；在非特别的中断结束情况下，最高优先级中断的正被服务位被自动地清除，而在特别的中断结束情况下，被清除的正被服务位却可以直接指定。无论这个正被服务位，是由中断响应置位的，还是由 CPU 直接写入正被服务寄存器中的该位置位的，这个正被服务位都将复位。若这个最高优先级中断的正被服务位被复位，则优先级屏蔽寄存器的位，将改变成反

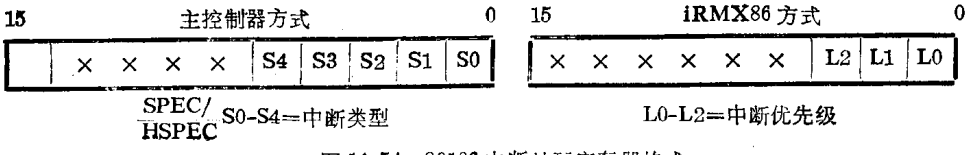


图 14.54　80186 中断结束寄存器格式

映要被服务的最低优先级中断。若一个低于最高优先级中断的正被服务位被复位，则优先级屏蔽寄存器的位将不被修改（因为正在被服务的最高优先级中断并没有改变）。在iRMX 86方式中，只支持特殊的EOI(End Of Interrupt，中断结束)这个寄存器是只写的。

七、中断状态寄存器

中断控制器，还含有一个中断状态寄存器(见图14.55)。这个寄存器含有4个有效位，其中，有3个位用来表示那一个计时器引起了一个中断。这种功能是必须的，因为在主控制器方式中，计时器是共享一个中断控制寄存器的。这3位中的某一位置位，就表示该位所对应的计时器产生了一个中断。在该计时器提出的中断请求被响应以后，寄存器中的相应位就被自动地清除。在同一个时刻，可能有多个位置位。中断状态寄存器中的第4位是DMA暂停位。当这一位置位时，就能阻止出现DMA操作。中断控制器一旦收到NMI，该位就被自动地置位。它也可以直接由程序员置位。每当执行了IRET指令，这一位就会被自动清除。在这个寄存器中的所有有效位，都是可以被读出和被写入的。

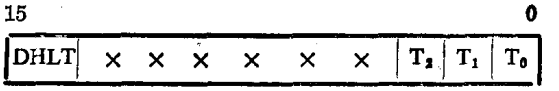


图14.55 80186中断状态寄存器格式

八、中断向量寄存器

在iRMX 86方式下，中断控制器还含有一个中断向量寄存器（见图14.56），用来指定一个中断类型向量的最高5位（这个中断类型的最低3位，由在iRMX 86方式下引起该中断的设备的优先级决定）。这个向量是作为对中断响应的回答，而放在CPU总线上的。

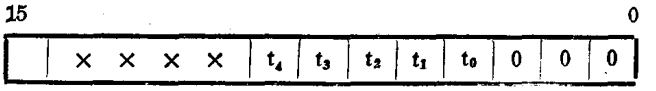


图14.56 80186中断向量寄存器格式(只存在于iRMX86方式中)

§14.6.4 中 断 源

80186中断控制器，接收来自内部和外部的许多中断源所提出的中断请求，并对它们进行仲裁。每一个中断源，在中断控制器中编程为具有不同的优先级。图14.57是产生一个中断请求的流程图，每个中断源都将根据这个流程，提出自己的中断请求。

一、内部中断源

内部中断源，是3个计时器和2个DMA通道。来自这些中断源的每一个中断，都被锁存入中断控制器，以便在引起中断的条件在原来集成的器件中已经消失以后，该内部中断请求，仍能在中断控制器中等待处理。等待处理的中断状态，通过读中断控制器中的中断请求寄存器而获得。对所有的内部中断来说，被锁存的中断请求，由处理器通过写中断请求寄存器而复位。要注意的是，在主控制器方式下，所有的计时器共享在中断请求寄存器中的一位，要知道究竟是哪一个计时器提出中断请求，可以把中断控制器的状态寄存器的值读出来进行判别。每

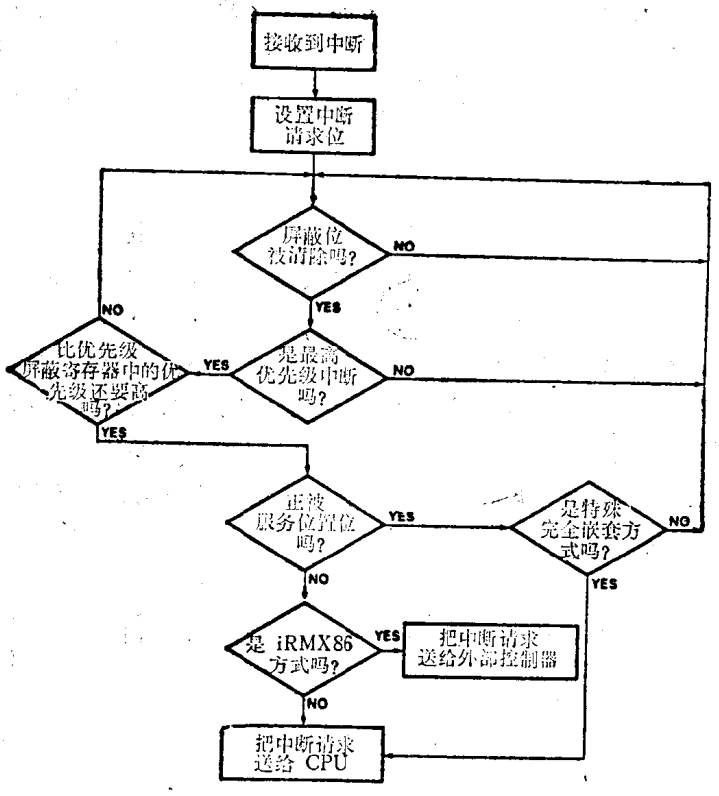


图 14.57　80186 中断请求序列

一个计时器有一个唯一的中断向量，因此，在中断服务程序中，就不需要再查询是哪个计时器提出了中断．同时，由于计时器是共享一个中断控制寄存器的，所以它们都被安排在同一个优先级．在它们之间，则又有着固定的优先级，其中计时器 0 优先级最高，计时器 1 次之，计时器 2 优先级最低．

二、外部中断源

80186 的中断控制器仅当它被编程为主控制器方式时，才接受外部的中断请求．在这种方式下，与中断控制器相连的外部引脚，可以作为直接的中断输入，或在程序选择下作为来自其他中断控制器的级联中断输入．这些选择，是通过对在 INT0 和 INT1 控制寄存器中的 C 和 SFNM 位，进行编程而作出的(见图 14.50)．

在被编程为直接中断输入时，4 个中断输入由其各自的中断控制寄存器控制．如前所述，这些寄存器含有为中断选择优先级的 3 个位、和使中断源能中断处理器的 1 个允许位．另外这些寄存器还含有一个位，它用来决定中断输入是边沿触发还是电平触发．在选择边沿触发方式时，产生一个中断以前，中断输入上必须出现一个低电平到高电平的变换；而在电平触发方式时，产生一个中断，只要在中断输入上保持高电平就可以了．在边沿触发方式下，输入在上升以前，必须至少保持一个时钟周期的低电平．两种方式相同的是，中断电平必须保持高电平，直到中断被响应为止，即中断请求没有被锁存入中断控制器．中断输入的状态，可以通过读中断请求寄存器来取得．每个外部引脚，都在这个寄存器中有相应的一位，以指示是否在某个引脚上出现了一个中断请求．必须注意，由于在这些输入上的中断请求，是不被中断控制器

锁存的，所以若外部输入变成不活跃，则该中断请求(和在中断请求寄存器中的相应位)也将变成不活跃(低电平)。同样，若中断输入是边沿触发方式，则在输入引脚上的低电平到高电平的变换，必须出现在中断请求寄存器中的中断请求位置位以前。

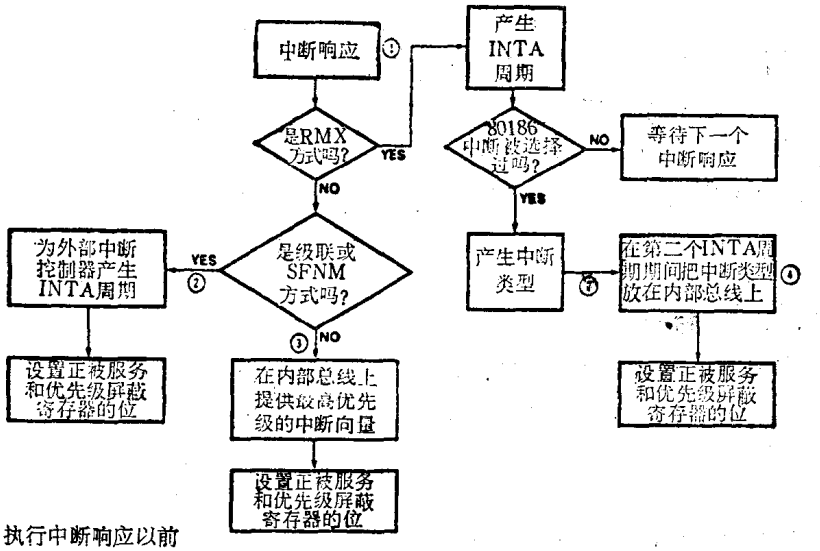
若 INT0 或 INT1 控制寄存器中的 C(Cascade) 位置位，则中断输入被级联到一个外部的中断控制器。在这种方式下，一旦出现在 INT0 或 INT1 引脚上的中断被响应，集成的中断控制器，将不为该中断提供中断类型。代替这一操作的，是将执行两个 INTA 总线周期，并把 INT2 和 INT3 引脚用来为 INT0 和 INT1 中断请求，提供中断响应脉冲。INT0/INT2 和 INT1/INT3，可以单独地编程入级联方式。这样，若使用了各有 9 个外部中断控制器的两个体，就可以有 128 个单独的向量化的中断源。

三、iRMX™86 方式中断源

当中断控制器以 iRMX86 方式构成时，集成的中断控制器，只接受来自集成的外围设备的中断请求，任何外部中断请求必须经过一个外部中断控制器。这个外部中断控制器，通过在 80186 上的 INT0 引脚，直接向 80186 提出中断服务请求。在这种方式下，这个引脚的功能不受集成的中断控制器影响。另外，在 iRMX86 方式中，集成的中断控制器，必须通过这个外部中断控制器，才能请求中断服务。这种中断请求是在 INT3 引脚上提出的。

§14.6.5 中断响应

80186 对一个中断，可以用两种不同的方式进行响应。在控制器处于主控制器方式、并且内部控制器提供中断向量信息的情况下，以第一种方式响应；若 CPU 从一个外部中断控制器中读出中断类型向量，或中断控制器是处于 iRMX86 方式的情况下，以第二种方式响应。在这两种场合下，由 80186 集成的中断控制器驱动的中断向量信息，是不能为 80186 微处理器以



1. 在 CPU 执行中断响应以前
2. 将要执行两个中断响应周期，中断类型由 CPU 在第二个周期准备就绪
3. 将不执行中断响应周期，中断向量地址被放在内部总线上，并不能为外部的处理器所使用
4. 在 iRMX86 方式下，中断类型不在外部总线上驱动

图 14.58 80186 中断响应序列

外的设备所使用的。

在每一种中断方式中，当集成的中断控制器，接收到一个中断时，该中断控制器就将自动地设置正被服务位和优先级屏蔽位，并把中断请求位复位。优先级屏蔽位，将防止控制器在具有较高优先级的中断服务程序被运行以前，又产生来自较低优先级的中断源的中断 CPU 的请求。另外，除非对应于某个中断的中断控制寄存器被设置为 SFNM (Special Fully Nested Mode)，中断控制器在相应于某一个中断引脚的正被服务位被清除以前，将阻止在该中断引脚上出现中断。图 14.58 是 80186 的中断响应序列。

一、内部导向、主控制器方式

在主控制器方式下，和所有的中断源相联系的中断类型是固定的，并且是不可改变的，这些中断类型在表 14.5 中给出。为了回答一个内部的 CPU 中断响应，中断控制器将产生一个向量地址，而不是中断类型。在 80186 (和 8086 一样)中，中断向量地址是中断类型乘以 4，这样加快了中断响应。

在主控制器方式下，集成的中断控制器，是系统的主中断控制器。这样安排的结果，外部的中断控制器不需要知道集成的控制器何时正在提供一个中断向量和中断响应正在行进；同时，不会产生中断响应总线周期。中断已被响应的第一个外部标志，是处理器从存贮器低区的中断向量表中读中断向量。

表 14.5 80186 中断向量类型

中断名	向量类型	缺省的优先级	中断名	向量类型	缺省的优先级
记时器 0	8	0	INT0	12	4
记时器 1	18	0	INT1	13	5
记时器 2	19	0	INT2	14	6
DMA0	10	2	INT3	15	7
DMA1	11	3			

由于不执行两个中断响应周期，并且不需要计算中断向量地址，所以对一个内部导向的中断的中断响应，只要 42 个时钟周期，它比需要外部导向的中断响应，或在 iRMX 86 方式下运行的中断控制器都要快。

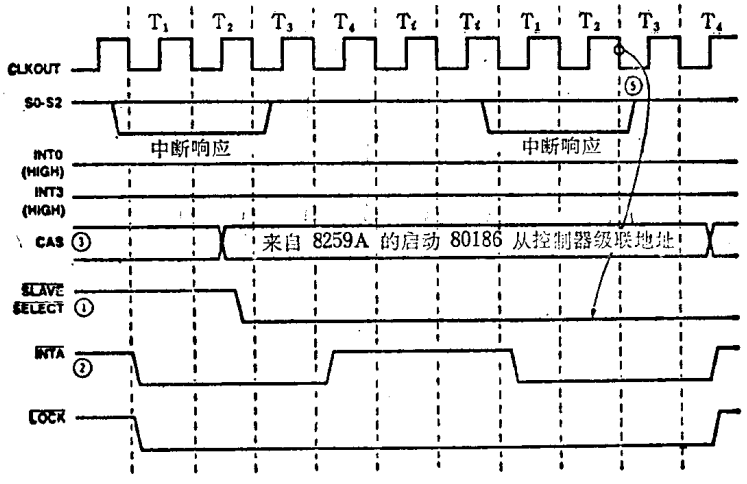
若发生了两个由程序指定的优先级相同的中断，则使用缺省的优先级规定 (如表 14.5 所示)。

二、内部导向、iRMX™ 86 方式

在 iRMX86 方式下，与各种中断源相联系的中断类型，是可改变的。中断类型的最高 5 位，取自中断向量寄存器，而最低 3 位取自引起中断的设备的优先级。由于在这种方式下中断控制器给出的是中断类型，而不是中断向量地址，所以在中断服务以前，CPU 必须计算出中断向量地址。

在 iRMX86 方式下，集成的中断控制器，将把中断类型送给 CPU，作为对 CPU 执行的两个中断响应总线周期的回答。在第一个中断响应总线周期期间，外部的主中断控制器，决定哪一个从中断控制器把它的中断向量放到微处理器的总线上；在第二个中断响应周期期间，处理器从它的总线上读取中断向量；因此这两个中断响应周期是必须被执行的，因为集成的中断控

制器仅在外部的中断控制器通知它具有最高的中断请求权时，才能送出中断类型信息（见图14.59）。80186 在第二个中断响应周期的 T_3 开始时的时钟下降沿期间，检查 SLAVE SELECT 信号。这个输入信号，必须在这个边沿以前的 20 ns 和以后的 10 ns 中保持稳定。



1. $\overline{\text{SLAVE SELECT}}$=INT1
2. $\overline{\text{INTA}}$=INT2
3. 由外部中断控制器驱动
4. $\overline{\text{SLAVE SELECT}}$ 必须在第二个 INTA 周期的 T_2 的节拍 2 之前驱动
5. 80186 读 $\overline{\text{SLAVE SELECT}}$

图 14.59　80186 iRMX86 方式中断响应定时

这两个中断响应周期，是背靠背地执行的，它们的执行将会因 $\overline{\text{LOCK}}$ 信号的活跃而被封锁（表示两个周期都被执行以后，才能响应 DMA 请求和 HOLD 请求）。两个中断响应周期，将总是被两个空闲 T 周期分开，并且当处理器总线接口尚未返回准备就绪信号时，在中断响应周期中，还要插入等待状态。插入两个空闲 T 状态，是为了与外部的 8259A 的定时相兼容。

由于在 iRMX86 方式下，必须运行中断响应周期，对内部产生的向量也不例外，更由于集成的中断控制器送出的是中断类型，而不是中断向量地址，所以在这种情况下，中断响应所需要的时间和外部导向的中断响应所需要的时间相同，即都是 55 个 CPU 时钟。

三、外 部 导 向

只要 80186 中断控制器是处于级联方式，特殊的完全嵌套方式或 iRMX86 方式（并且集成的中断控制器不是由外部主中断控制器所启动的），外部中断导向就会发生。在这种方式下，80186 产生两个中断响应周期，并在第二个中断响应周期从地址数据总线的低 8 位读出中断类型（见图 60）。这种中断响应和 8086 的中断响应完全相同，所以 8259A 中断控制器也可以和在 8086 系统中一样地使用。要注意的是，两个中断响应周期都是封锁的，并且在两个中断响应总线周期之间，总是插入两个空闲 T 状态，在处理器尚未收到准备就绪信号时，还要在中断响应周期中插入等待状态。还要注意的是，80186 提供了两个中断响应信号，一个提供给在 INT0 引脚上出现的中断，另一个提供给在 INT1 引脚上出现的中断。这两个中断响应信号是互斥的。在 INT2/$\overline{\text{INTA}}$0 或 INT3/$\overline{\text{INTA}}$1 标志一个中断响应时，中断响应状态将在状态引脚（$\overline{S}_0$—$\overline{S}_2$）上得到反映。

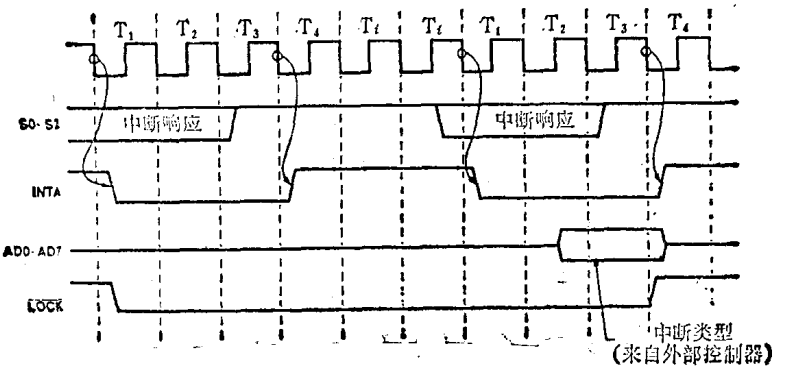


图 14.60　80186 级联中断响应定时

§14.6.6　中断控制器的外部连接

四个中断信号，可以被程序地构成 3 种主要的选择。它们是：直接中断输入（用集成的控制器提供中断向量），级联（用一个外部中断控制器提供中断向量）以及 iRMX 86 方式。在所有这些方式中，在外部引脚上出现的中断请求必须保持有效，直到该中断被响应为止。

一、直接输入方式

当级联方式位被清除时，中断输入引脚，就被构成直接中断输入引脚（见图 14.61）。在这种方式下，一个中断源（例如 8272 软磁盘控制器）可以被直接连接到中断输入引脚。一旦在这个输入引脚上接收到一个中断，并且这个中断是被允许的，它的优先级也是最高的（否则集成的中断控制器将不做任何操作），这时集成的中断处理器就把这个中断发送给 CPU，并等待中断响应。当中断响应出现时，它将把中断向量地址送给 CPU。在这种方式下，CPU 将不执行任何中断响应周期。

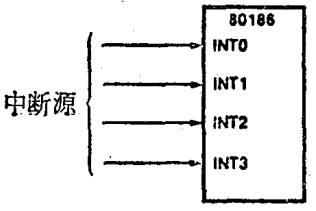


图 14.61　80186 非级联的中断连接

这些引脚可以用各自的控制寄存器，单独编程为边沿触发方式，或电平触发方式。在边沿触发方式下，在输入引脚上必须要有一个低电平到高电平的变换，才能产生对 CPU 的中断请求；在电平触发方式下，只要在输入引脚上出现高电平，就能产生一个中断。在边沿触发方式下，中断输入必须保持低电平至少一个 CPU 时钟周期，以便确保能被识别。在两种方式中，中断输入都必须保持活跃，直到被响应为止。

二、级联方式

当级联方式位被置位（C = 1），并且 SFNM 位被清除时，中断输入引脚处于级联方式。这时，中断输入引脚和中断响应引脚是成对的。INT2/$\overline{\text{INTA0}}$ 和 INT3/$\overline{\text{INTA1}}$ 引脚具有双重的用途：一是起直接输入引脚的作用；二是起中断响应输出引脚的作用。INT2/$\overline{\text{INTA0}}$ 为 INT0 输入提供中断响应，而 INT3/$\overline{\text{INTA1}}$ 则为 INT1 输入提供中断响应。图 14.62 表示了这种连接。

在编程为这种方式时，80186要提供两个中断响应脉冲，来响应在 INT0 引脚上出现的中断。这些脉冲是在 INT2/$\overline{\text{INTA}}$0 引脚上提供的，并且将在$\overline{S_0}$—$\overline{S_2}$引脚上产生的中断响应状态中得到反映。中断类型将在第二个脉冲时读入。

INT0/INT2/$\overline{\text{INTA}}$0和INT1/INT3/$\overline{\text{INTA}}$1，可被单独地编程为中断请求/响应对，或被编程为直接输入。这表示 INT0/INT2/$\overline{\text{INTA}}$0 可被编程为一个请求/响应对，而同时 INT1 和 INT3/INTA1 则分别被编程为提供内部导向的中断输入。

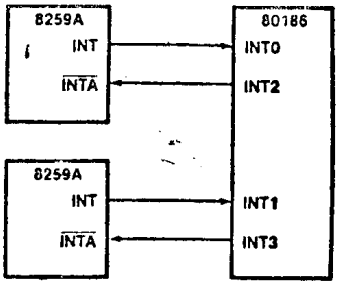


图 14.62 80186 级联和特殊的完全嵌套方式接口

当在级联方式下收到一个中断时，在专用的中断控制寄存器中的优先级屏蔽位、和正被服务位，将被置位入中断控制器的正被服务和优先级屏蔽寄存器。这样，就能防止中断控制器产生一个较低优先级的 80186 CPU 中断请求。同样，由于正被服务位被置位，所以在该特定的中断输入引脚上的后续中断请求，将不会使集成的中断控制器再产生对 80186 CPU 的中断请求。这说明，当外部中断控制器，在它的某一个输入引脚上，接收到一个较高优先级的中断请求，并把它送到 80186 的中断请求引脚上以后，集成的中断控制器要在它内部的对应于该中断输入引脚的正被服务位被清除以后，才会把该信号继续送给 80186 CPU。

三、特殊的完全嵌套方式

在级联方式位和 SFNM 位都置位时，中断输入引脚，被构成特殊的完全嵌套方式。这种方式的外部接口和级联方式完全一样。唯一不同的是，使中断能从外部中断控制器送到集成的中断控制器，从而使中断 80186 CPU 的条件不一样。

当一个中断被从处于特殊的完全嵌套方式（简称 SFNM 方式）下的中断引脚上接收到时，若该中断的优先级是最高的，则不管在中断控制器中对应于该中断源的正被服务位处于什么状态，CPU 将被这个请求所中断。当一个中断，被从一个 SFNM 方式的中断引脚上响应时，在专用的中断控制寄存器中的优先级屏蔽位和正被服务位将被置位入中断控制器的正被服务和优先级屏蔽寄存器。这样就能防止中断控制器，产生一个较低优先级的中断 80186 CPU 的中断请求。但是，和级联方式不一样，中断控制器将不阻止由同一个外部中断控制器，产生另外的中断 80186 CPU 的请求。这说明，当外部中断控制器在它的一个引脚上接收到一个较高优先级的中断，并把它送到集成的中断控制器的中断请求引脚以后，不管该中断引脚的正被服务位处于什么状态，它将产生一个 80186 CPU 中断。

若 SFNM 方式位置位，而级联方式位并不置位，控制器将提供内部中断导向，同时也将忽略用来决定是否把中断请求发送给 CPU 的正被服务位的状态。换句话说，它将使用产生中断的 SFNM 条件，并使用内部导向来进行中断响应，即，若到来的中断，是具有最高优先级的中断，则不管对应于该中断的正被服务位处于什么状态，它都将引起 CPU 的中断。

四、iRMX86 方式

当外围再定位寄存器中的 RMX 位置位时，中断控制器被设置成 iRMX 86 方式。在这种方式下，中断控制器的所有四个输入引脚都用来与外部主中断控制器进行信号交换。图 14.63

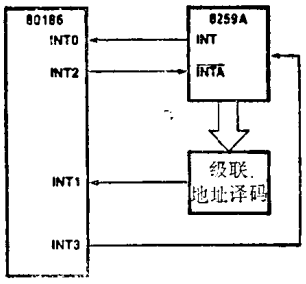


图 14.63　80186 iRMX86 方式接口

表示了在 iRMX 86 方式下，80186 中断引脚和一个外部中断控制器的硬件之间的连接。

由于集成的中断控制器，是一个从中断控制器，所以它必须能为一个外部中断控制器产生中断输入。它也必须知道，当它有最高优先级的中断请求时，何时才能把它的中断向量放到总线上去。这两个信号分别是由 INT3/Slave Interrupt Output 和 INT1/Slave Select 引脚提供的。外部主中断控制器必须能中断 80186 CPU，并且需要知道中断请求何时被响应。INT0 和 INT2/$\overline{\text{INTA0}}$ 引脚分别提供这两种功能。

§14.6.7　8259A/级联方式接口举例

图 14.64，是 80186 和 8259A 在级联中断方式下接口的例子。启动 80186 中断控制器的程序，在 §14.11.5 中给出。注意，一个“中断准备就绪”信号必须返回给 80186，以防止产生等待状态来回答中断响应周期。在这种结构中，INT0 和 INT2 引脚用作直接中断输入。因此，这种结构提供了 10 个外部中断引脚，其中 2 个是由 80186 中断控制器本身提供的，还有 8 个是外部的 8259 A 提供的。同样，8259 A 在这种结构中也是主中断控制器，它将只接收对它产生的中断进行回答的中断响应脉冲。它可以再次和最多可达 8 个的 8259 A 级联(这些 8259 A 都是处于从中断控制器方式)。

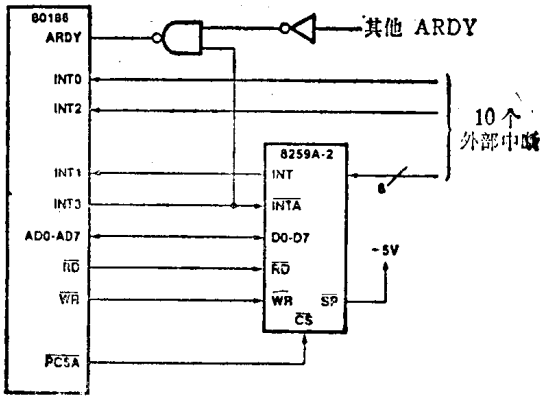


图 14.64　80186/8259 A 中断级联

§14.6.8　80130 iRMX™ 86 方式接口举例

图 14.65，是在 iRMX 86 方式下 80186 和 80130 相接口的例子。在这种方式下，80130 的中断控制器，是系统的主中断控制器。当某一个 80186 的集成的外围部件形成了一个中断条件，并且该条件能满足 80186 集成的中断控制器产生一个中断的要求时，80186 就向 80130 的中断控制器提出一个中断请求。要注意，80130 是直接从 80186 的状态信号中译码出中断响应状态的，因此，80186 的 INT2/$\overline{\text{INTA0}}$ 引脚，不需要连接到 80130。图 14.65 中使用这个中断响应信号，来启动级联地址译码器。在第二个中断响应周期的 T_1 期间，80130 在 AD_8—AD_{10} 上

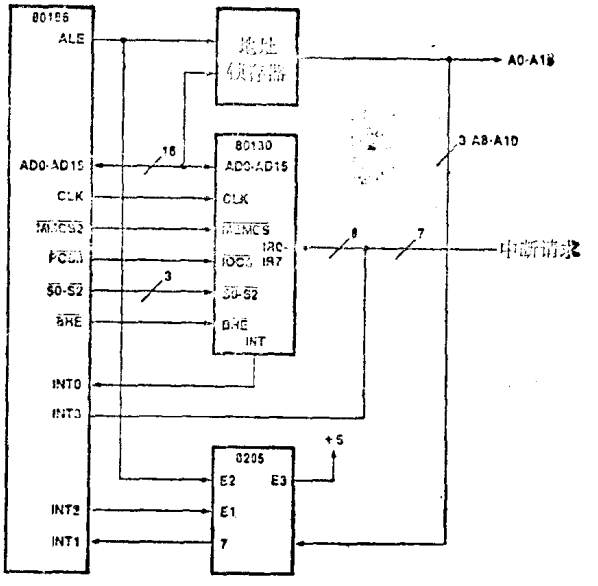


图 14.65　80186/80130 iRMX 86 方式接口

驱动级联地址。这个级联地址，被锁存入系统地址锁存器，并且若 8205 译码器译出恰当的级联地址，则 80186 的 INT1/$\overline{\text{SLAVE SELECT}}$ 信号，将被驱动为活跃，从而启动 80186 集成的中断控制器，把它的中断向量放在内部总线上。使 80186 进入 iRMX86 方式的程序在 §14.11.5 中。

§14.6.9　中断等待时间

中断等待时间，是指从 80186 收到中断请求，到它开始响应该中断请求的时间。这个时间不同于中断响应时间。中断响应时间，是从处理器真正开始处理某一中断，到真正执行该中断的服务程序的第一条指令的时间。影响中断等待时间的因素，是正在执行的指令和中断允许触发器。

因为仅当在 CPU 中的中断允许触发器置位时，中断才会被响应，所以，当中断不被处理器所允许时，中断等待时间就会很长。

当 CPU 允许中断时，中断等待时间就是正被执行的指令的函数。只有重复执行的指令，才能在它们没有执行完以前的两次重复之间中断。这表示中断等待时间最长可达 69 个 CPU 时钟周期，这个时间正是处理器用来执行 80186 中最长的整除指令（带有一个段超越前缀）所需要的时间。

其他的因素，也能影响中断等待时间。在执行一个前缀（如段超越前缀和封锁前缀）和执行前缀所属的指令之间，是不允许发生中断的。另外在一条修改任何段寄存器的指令和紧接在这条指令之后的一条指令之间，也是不允许发生中断的。这种限制是改变堆栈操作所需要的。若在这种操作过程中发生了中断，则从中断返回的地址，就将放在一个无效的堆栈中（因为堆栈段寄存器已经被修改，而堆栈指示器还没有被修改）。最后，当 $\overline{\text{TEST}}$ 输入活跃时，在执行 WAIT 指令和紧接着的下一条指令之间，也是不允许发生中断的。但是若在执行 WAIT 指令期间，$\overline{\text{TEST}}$ 输入是不活跃的，则中断将被接受，并在中断返回以后，继续执行 WAIT 指令。

因为 WAIT 指令，是用来在 8087 正处于忙状态时，防止 80186 执行 8087 指令的，所以这种规定也是必需的。

§14.7 时钟发生器

80186 包含有一个时钟发生器，它为所有 80186 的集成部件，和 80186 系统中的所有 CPU 同步的设备产生主时钟信号。这个时钟发生器包括晶体振荡器、除以 2 计数器、复位电路和准备就绪信号发生逻辑。图 14.66 是该时钟发生器的方框图。

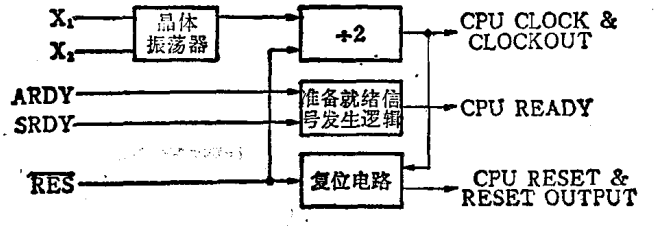


图 14.66 80186 时钟发生器方框图

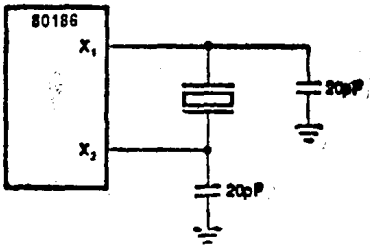


图 14 67 80186 的晶体连接

§14.7.1 晶体振荡器

80186 晶体振荡器，是一个并联振荡器。它的使用方法如图 14.67 所示。所示的电容值，是个约数。在晶体频率下降时，它们的值应该增大，以便在 80186 提供 4 MHz 的最小晶体频率时，可以取值 30 pF。这个振荡器的输出，在 80186 以外是不可直接使用的。

§14.7.2 使用外部振荡器

80186 也可以和一个外部时钟发生器一起使用。外部频率输入信号 EFI(External Frequency Input)，应该直接连接到 80186 振荡器的 X_1 输入引脚，而 X_2 在这种情况下应该空着。这个振荡器输入，用来驱动一个内部的除以 2 计数器，以便产生 CPU 时钟信号。因此，只要最小高电平时间和最小低电平时间得到满足，外部频率输入实际上可以具有任意的占空周期。

§14.7.3 时钟发生器

晶体振荡器(或外部频率输入)驱动一个除以 2 电路，并由该电路为 80186 系统，产生一个具有 50% 占空周期的时钟。所有的 80186 定时，都是以这个时钟作为标准的，这个时钟的信号在 80186 的 CLKOUT 引脚上可供使用。这个信号在 EFI 信号从高电平到低电平的变换时改变状态。

§14.7.4 产生准备就绪信号

时钟发生器还包括了产生准备就绪信号的电路，同 SRDY 和 ARDY 输入的接口可参见

§14.3.1 的六。

§14.7.5. 复　　位

80186 时钟发生器，还为系统提供了一个同步的复位信号。这个信号产生于对 80186 的复位输入(RES)，该信号同步于时钟输出(CLKOUT)信号。

复位输入信号，也把除以 2 计数器复位。在 RES 输入信号首先变活跃以后，就会产生一个 1 个时钟周期的内部清除脉冲。该脉冲在 RES 变活跃以后的 X_1 输入引脚上第一个低电平到高电平变换时开始变活跃，并且在 X_1 输入引脚上的下一个低电平到高电平的变换时变得不活跃。为了确保在下一个 EFI 周期产生清除脉冲，RES 输入信号，对高电平到低电平的 EFI 输入信号，必须满足 25 ns 的建立时间(见图 14.68)。在这个清除过程中，时钟输出将是高电平。在 X_1 的下一个高电平到低电平变换时，时钟输出将会变成低电平，并且会在 EFI 的每一个后续高电平到低电平的变换时改变状态。

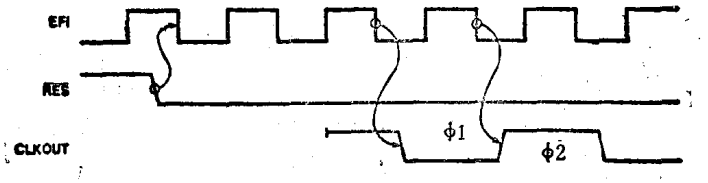


图 14.68　80186 时钟发生器复位

这个复位信号，将传送给 80186 的其余部分，同时，在 80186 RESET 输出引脚上出现的信号，被 80186 的时钟输出信号的高电平到低电平的变换所同步。只要 RES 输入保持活跃，这个复位信号就保持有效。在 RES 输入变得不活跃以后，经过 6.5 个 CPU 时钟周期(即取第一条指令的 T_1，将在 6.5 个时钟周期以后出现)，80186 将开始取第一条指令(在存贮器单元 FFFF0H)。为了确保 RESET 输出在下一个 CPU 时钟周期变成不活跃，RES 输入变不活跃的边沿和 80186 时钟输出信号的低电平到高电平变换的边沿之间，要满足一定的保持和建立时间(见图 14.69)。

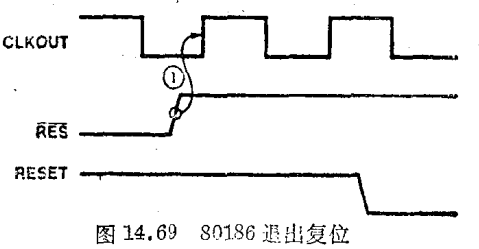


图 14.69　80186 退出复位

§14.8 片　　选

80186 含有一个片选部件，它为由 80186 CPU 和 DMA 部件产生的存贮器和 I/O 访问，产生硬件片选信号。这个部件是可程序的，因此它可以满足大多数小的和中等容量的 80186 系统的片选要求(按照存贮器设备或体的容量和速度)。

片选信号，是只为内部产生的总线周期而驱动的。任何由外部部件(例如，外部 DMA 控制器)产生的总线周期，将不会使片选信号活跃。因此，任何外部总线主设备，必须负责产生自己的片选信号。由于在一个总线 HOLD 期间，80186 并不浮空它的片选信号，因此，必须有逻辑电路，来启动那些外部总线主设备希望访问的设备(见图 14.70)。

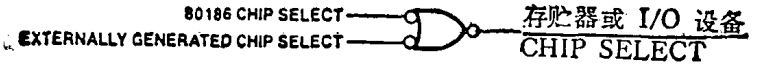


图 14.70 80186/外部片选/设备片选信号的产生

§14.8.1 存贮器片选

80186 提供了 6 个相分离的片选引脚，用来在一个 80186 系统中连接存贮器器件。在这些引脚上的信号，被命名为 $\overline{UCS}$、$\overline{LCS}$ 和 $\overline{MCS_{0-3}}$。其含义分别为高部存贮器片选(Upper Memory Chip Select)、低部存贮器片选(Lower Memory Chip Select)和中部存贮器片选 0—3(Midrange Memory Chip Selects 0—3)，被用来(但不限于)和 80186 系统存贮器的三个主要区域相联系(见图 14.71)

图 14.71 80186 存贮器区域和片选

从这些信号的名字就可以猜出，高部存贮器、低部存贮器和中部存贮器片选信号，是用来寻址在 80186 系统中存贮器的高部、低部和中部区域的。$\overline{UCS}$ 的限和 LCS 的限各自固定在存贮器空间的 FFFFFH 和 00000H。这些区域的其他限，则要根据存贮器的容量，对片选信号的控制寄存器进行编程来设定。对中部存贮器，每个中部存贮器区域的基地址和存贮器块的容量都可以进行编程，唯一的限制是基地址必须被编程为整个块容量的整数倍。例如，若块容量是 128K 字节(4 个 32K 字节块)，则基地址可以是 0 或 20000H，但不能是 10000H。

存贮器片选信号，是由在外围控制块中的 4 个寄存器控制的(见图 14.72)。$\overline{UCS}$ 和 $\overline{LCS}$ 信号，各使用一个寄存器。这两个寄存器的值，是所对应的信号寻址的存贮器块的容量。另外两个寄存器，用来控制中部存贮器块的容量和基地址。

偏移量：

A0H	高部存贮器容量	①	UMCS
A2H	低部存贮器容量	②	LMCS
A4H	外围片选基地址	③	PACS
A6H	中部存贮器基地址	④	MMCS
A8H	中部存贮器容量 EX MS	⑤	MPCS

⑥

1. 高部存贮器准备就绪位
2. 低部存贮器准备就绪位
3. PCS0—PCS3 准备就绪位
4. 中部存贮器准备就绪位
5. PCS4—PCS6 准备就绪位
6. MS：1=外围设备在存贮器空间活跃　0=外围设备在 I/O 空间活跃
   EX：1=7 个 PCS 信号　0=PCS5=A1，PCS6=A2

注：并不是每个字段的所有位都使用

图 14.72 80186 片选控制寄存器

在复位时，只有 $\overline{UCS}$ 是活跃的。它被编程为在复位时活跃最高的 1K 存贮器块，给所有的存贮器取操作插入 3 个等待状态，并成为每个存贮器取操作的外部准备就绪信号的一个因素。所有其他的片选寄存器在复位以后状态都是不确定的，一直要到对应一个信号的所有必

要的寄存器都被访问过以后，才会有信号活跃(不一定要写，对一个未初始化的寄存器的读，就能启动由该寄存器控制的片选功能)。

§14.8.2 外围片选

80186提供了7个相分离的片选引脚，用来连接在80186系统中的外围器件。这些信号被命名为 $\overline{PCS_{0-6}}$。这些信号中的每一个对应在一个被编程指定的基地址以上的存贮器、或I/O空间中的七个相连的各为128个字节的区域中的一个区域。

外围片选信号，是由在内部外围控制块中的两个寄存器控制的(见图14.72)。这些寄存器使得外围器件的基地址能被设置，并且使外围器件能够映照到存贮器或I/O空间。在任何外围片选信号变活跃以前，这两个寄存器都必须被访问过。

在MPCS寄存器中的一个位，使得 $\overline{PCS_5}$ 和 $\overline{PCS_6}$ 变成被锁存的 A_1 和 A_2 输出。在作出这种选择时，$\overline{PCS_5}$ 和 $\overline{PCS_6}$ 将在整个总线周期中，反映 A_1 和 A_2 的状态。提供这些信号，是为了使外部的外围寄存器选择，在一个地址不锁存的系统中得以进行。在复位时，这些信号被驱动为高电平。在PACS和MPCS都被访问(并被编程为提供 A_1 和 A_2!)以后，它们将只反映 A_1 和 A_2 的状态。

§14.8.3 准备就绪信号的产生

80186包含了一个产生准备就绪信号的部件，为对存贮器或I/O区域的所有访问，产生一个内部准备就绪信号，对这个信号80186的片选电路将作出响应。

对每个准备就绪信号产生区域，该内部部件可以插入0—3个等待状态。表14.6表示了准备就绪控制位，应该怎样被编程来提供这些等待状态。另外，准备就绪信号产生电路，可以被编程为忽略外部的准备就绪信号的状态(即只使用内部的准备就绪信号电路)，或成为外部准备就绪信号状态的一个因素(即，仅在内部的准备就绪信号电路和外部的准备就绪信号电路都准备就绪以后，准备就绪信号才会发送给处理器)。但是在该部件中，还必须包含某种能产生一个外部准备就绪信号的电路，因为在复位时，准备就绪信号发生器，被编程为所有对最高1K存贮器块进行访问的外部准备就绪信号的一个因素。若没有一个准备就绪信号在任何一个外部准备就绪引脚(ARDY或SRDY)上返回，则处理器将一直等着去取第一条指令。

表14.6 80186等待状态编码

R_2	R_1	R_0	等待状态数
0	0	0	0+外部准备就绪信号
0	0	1	1+外部准备就绪信号
0	1	0	2+外部准备就绪信号
0	1	1	3+外部准备就绪信号
1	0	0	0(不需要外部准备就绪信号)
1	0	1	1(不需要外部准备就绪信号)
1	1	0	2(不需要外部准备就绪信号)
1	1	1	3(不需要外部准备就绪信号)

§14.8.4 片选用法举例

在本章描述总线接口的§14.3.2中，举了不少使用片选信号的例子。这些例子说明使用80186提供的片选信号，是非常简单的，要记住的关键是，这些片选信号只在由80186 CPU或DMA部件产生的总线周期期间才会被激活。当另一个总线主设备占有总线时，它必须产生它自己的片选信号。此外，在80186把总线交给其他总线主设备时（通过HOLD/HLDA），80186并不浮空这些片选信号。

§14.8.5 重叠的片选区域

一般来说，80186的片选信号，不应该编程为使任意两个被选的区域相重叠。另外，也不能把片选区域编程为和集成的256字节的控制寄存器块的任何单元所重叠。出现重叠的结果是：

只要两个片选信号被编程为指定同一个区域，则在对该区域所进行的任何访问期间，两个片选信号都会被激活。在这种情况下，这两个片选信号控制寄存器中关于该区域的准备就绪位，就应该具有相同的值。若不是这样，处理器对在这个区域进行访问的响应，将是不确定的。

若任何片选区域和集成的256字节控制寄存器块相重叠，则片选信号的定时，就会被改变。一般从这次访问中，在外部总线上返回的值将被忽略。

§14.9 在80186系统中的软件

由于80186同8086和8088是目标代码相兼容的，所以在80186系统中的软件，就和在8086系统中的软件十分相似。但是，由于80186具有硬件片选功能，所以若要使用这种功能，80186系统中就应该含有一定数量的初始化程序。

§14.9.1 80186系统初始化

一个计算机系统的许多可程序的部件，在被使用以前，都必须初始化，80186系统也不例外。80186包含有直接影响寻址存贮器和I/O空间的系统能力的电路，即片选电路。这个电路，必须在存贮器区域和外围设备被使用片选信号寻址以前初始化。

在复位时，UMCS寄存器被编程为对在存贮器最高1K字节空间进行的所有存贮器取操作活跃。它也被编程为给所有在这个空间中的存贮器访问，插入3个等待状态。若要使用硬件片选信号，则这些信号必须在处理器离开这1K存贮器区域以前被编程。若出现一个向并没有被选中的片子的跳转，则微处理器系统就将停止操作（因为处理器将要从数据总线上取无用数据）。§14.11.6中提供了一个典型的80186片选部件的初始化程序。

一旦片选部件被初始化以后，80186的其余部分的初始化，就和8086系统基本相同。例如，设置中断向量表，把中断控制器初始化，把一个串行I/O通道初始化，然后就开始执行主程序。要注意，80186中集成的外围部件，并不共享在8086系统中实现这些功能的标准的Intel

外围部件所使用的程序模块，即不同的值必须被编程入不同的寄存器，以便在使用集成的外围部件时，取得相同的功能。§14.11.6 提供了使用 80186 中断控制器的一个中断驱动系统的初始化程序。

§14.9.2 iRMX™ 86 系统初始化

80186 和 iRMX 86 操作系统一起使用需要一个外部的 8259A 和一个外部的 8253/4，或一个外部的 80130OSF 器件。需要这些器件，是因为操作系统是被中断所驱动的，因而希望中断控制器和计时器具有这些外部器件的寄存器模块。这个模块并不就是 80186 提供的模块。因此，在复位以后，80186 必须进入 iRMX 86 方式。这一步初始化操作，可以在复位以后，跳转到 iRMX 86 系统的根任务之前的任何时刻执行。在需要的时候，一小段初始化 80186 片选和 80186 中断控制器程序，可以插在复位向量单元和 iRMX 86 系统的起始位置之间（见图 14.73）。在这种情况下，复位时，处理器将会跳转到 80186 的初始化程序，并且在这项初始化工作完成以后，再跳转到 iRMX 86 的初始化程序（在根任务中）。在 iRMX 86 操作开始以前，把 80186 的硬件初始化很重要，因为若以相反的次序进行初始化，iRMX86 系统，可能不能正确地对某些由 80186 寻址的资源进行初始化。

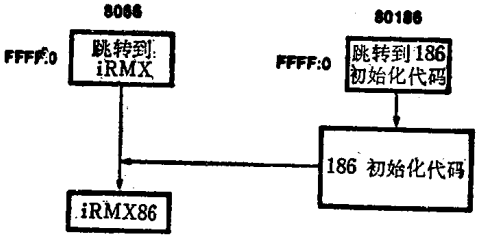


图 14.73 8086 和 80186 初始化 iRMX86

§14.9.3 8086 和 80186 执行指令的区别

8086 和 80186，在有些指令的执行上的区别是：

1. 无定义的操作码

当操作码 63H、64H、65H、66H、67H、F1H、FEH XX111XXXB 和 FFH XX111XXXB 被执行时，80186 将会执行非法指令异常，它的中断类型是 6，而 8086 将忽略这些操作码。

2. 0FH 操作码

在遇到 0FH 操作码时，8086 将执行 POP CS，而 80186 将执行一个中断类型为 6 的非法指令异常。

3. 在偏移量 0FFFFH 执行字写操作

在某个段的偏移量 0FFFFH 执行一个字写操作时，8086 将在偏移量 0FFFFH 写入一个字节，并把另一个字节写入偏移量 0，而 80186 则将在偏移量 0FFFFH 写入一个字节，并把另一个字节写入偏移量 10000H（在该段以外的一个字节）。若在堆栈指示器含有值 1 时，执行堆栈 PUSH 操作，同样也会发生一字节段溢出（仅对 80186）。

4. 用大于 31 的值进行移位/环移

80186 在用一个值进行移位、或环移操作前（用 CL 或一个立即数值），将把该值和 1FH 相与，从而把移位和环移的次数，限制在比 32 小的范围之内。8086 是不执行这一步操作的。

5. LOCK 前缀

8086 在执行 LOCK 前缀以后，立即激活它的 LOCK 信号，80186 则要到处理器准备开始

执行具有该 LOCK 前缀的指令的数据周期时，才激活 LOCK 信号。

6．被中断的串传送指令

若 8086 在执行一条重复的串传送指令时被中断，它推入堆栈的返回值，将指示着该串传送指令之前的最后一个前缀。若该指令具有多个前缀(例如除重复前缀外，还有一个段超越前缀)，则在从中断返回时，它将不会再被执行。对 80186 来说只要前缀不是重复的，它将把该重复指令的第一个前缀的地址推入堆栈，从而使该指令在中断返回以后，仍能正常地重新执行。

7．使用整数除法产生除法错误的条件

8086 只要商的绝对值大于 7FFFH (对字操作)，或大于 7FH (对字节操作)，就会产生一个除法错误；80186 扩展了负数的范围，使商能包括 8000H 和 80H。这些数表示了使用 2 的补码所能表示的最小负数(各自相当于十进制 - 32768 和 - 128)。

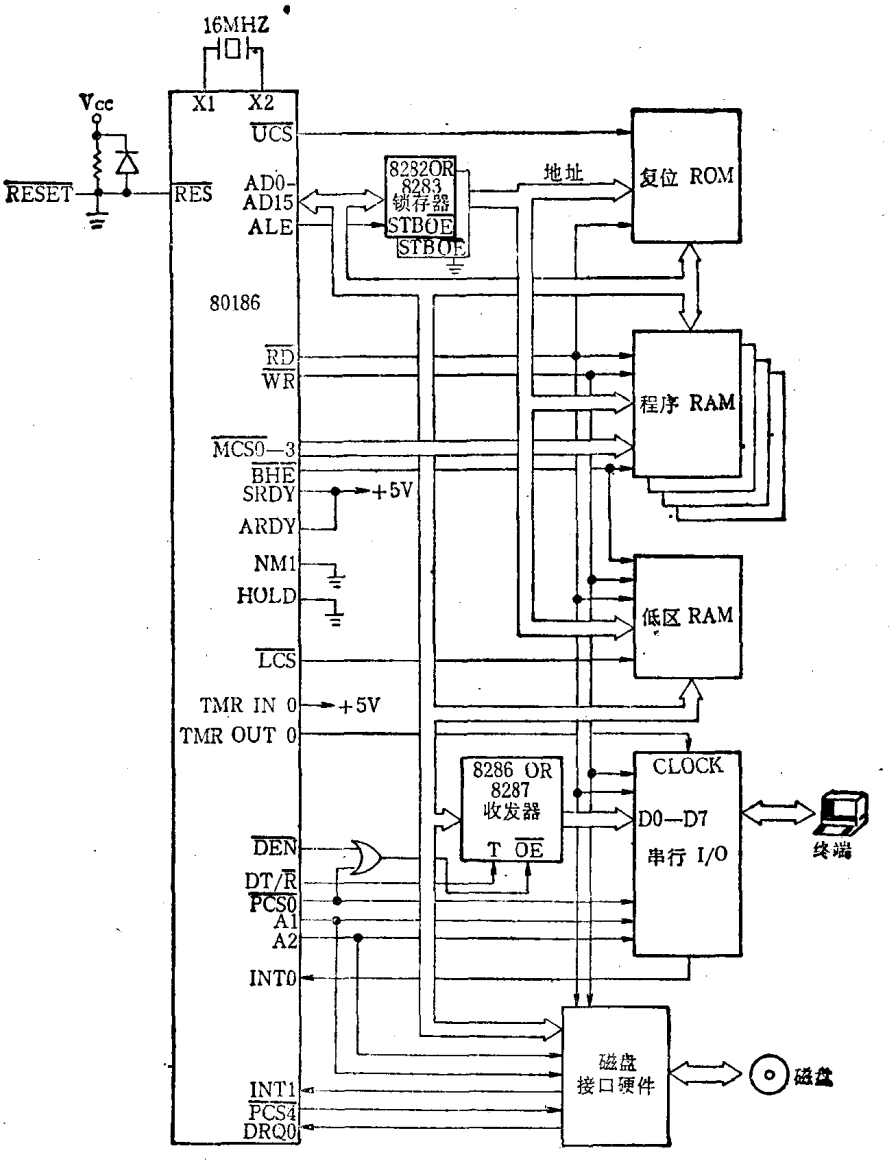


图 14.74　典型的 iAPX 186 计算机

8. ESC 操作码

80186 可以编程为每当执行一条 ESC 指令，就引起一个中类型为 7 的中断；8086 没有这种规定。80186 在执行这种自陷功能前，必须是已被对这种操作编程的。

这些区别可以用来判断一个程序是正在 8086 上执行，还是在 80186 上执行。要实现这种判断最有效的方法，是观察多位移位指令的执行。例如，若指定多位移位的计数是 33（存放在 CL 寄存器中！），8086 将执行移位 33 次，而 80186 将只移位一次。

除了上面提到的指令执行上的区别以外，80186 还包含了一些新的指令类型，它们简化了处理器的汇编语言程序，并且提高了处理器执行高级语言的性能。§ 14.11.8 中介绍了这些指令。

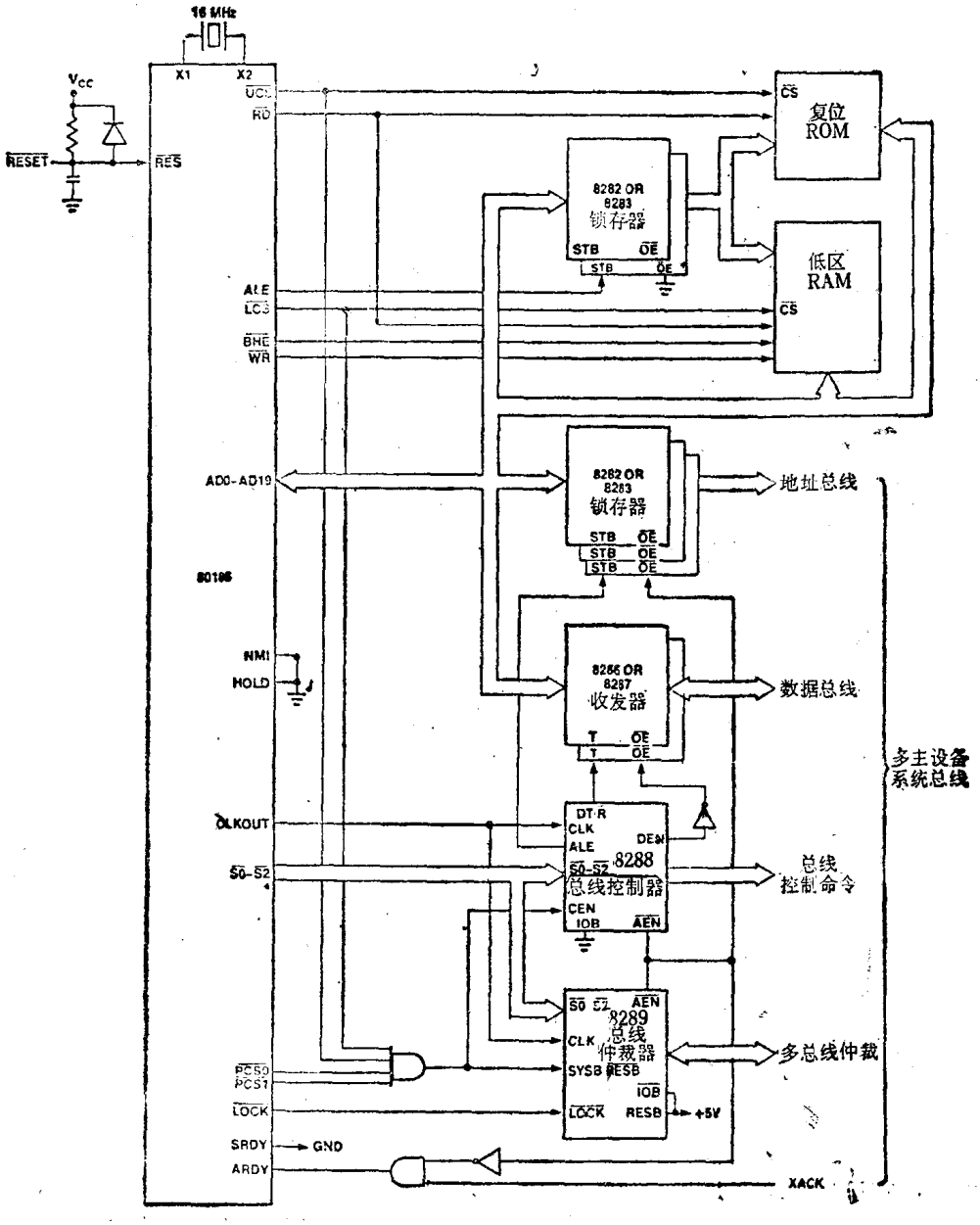


图 14.75　典型的 iAPX 186 多主设备总线接口

§14.10 结　论

80186是集成电路技术高度发展的产物，由于它在一个芯片上集成了许多必须的外围电路和接口，所以系统设计的任务就被大大简化，同时整个系统的体积也大大缩小。由于减少了大量的外部连接，系统的可靠性得到了较大的提高。因此用80186设计的系统的性能，比体积和价格相仿的其他系统的性能要高得多。

图14.74是典型的iAPX 186计算机的结构框图，而图14.75则是典型的iAPX 186多主设备总线接口框图。

§14.11 80186技术资料

§14.11.1 外围控制块

80186中所有集成的外围器件，都是由在一个集成的外围控制块中的寄存器集控制的。从物理位置上说，是处于它们所控制的外围器件中，但在寻址时，却作为一个寄存器块来处理。这个寄存器集有连续的256个字节，并且可以放在80186的存贮器、或I/O空间的以256个字节作为边界的任何地方(即起始地址必须是256的整数倍)。图14.76表示了这些寄存器。

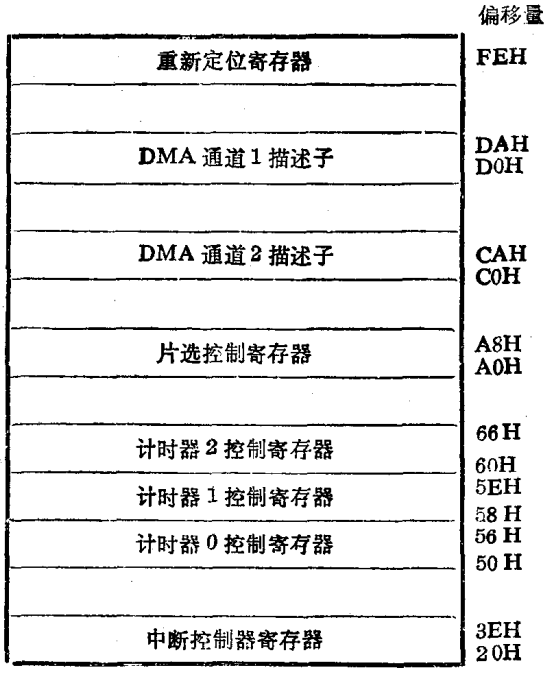
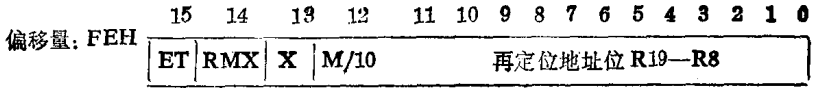


图14.76　80186集成的外围控制块

一、设置外围控制块的基地址

外围控制块，除了各种80186外围器件的控制寄存器以外，还有一个外围控制块再定位寄存器。这个寄存器使该外围控制块，能在处理器的存贮器或I/O空间中，以256字节为界进行



ET=1: ESC 自陷
=0: 无 ESC 自陷
M/IO=1: 寄存器块定位在存贮器空间中
=0: 寄存器块定位在 I/O 空间中
RMX=0: 主中断控制器方式
=1: 与 iRMX 兼容的中断控制器方式

图 14.77 80186 重新定位寄存器方式

浮动。图 14.77 是这个寄存器的格式。

这个寄存器在外围控制块中的偏移量是 FEH。由于它本身也包含在该外围控制块中，所以一旦外围控制块的位置移动了，该重新定位寄存器的位置也将随之而移动。

除了外围控制块重新定位信息以外，重新定位寄存器还含有两个附加的位。一个用来把中断控制器设置为处于 iRMX 86 兼容方式；另一位置位时，使处理器一遇到 ESC 指令就自陷。

由于重新定位寄存器是处于外围控制块中的，所以在复位时，该重新定位寄存器，被自动地编程为具有值 20FFH。这表示外围控制块将被放在 I/O 空间的最高区域 (FF00H 到 FFFFH)。因此，在复位以后，重新定位寄存器将被放在 I/O 空间的字单元 FFFEH。

若用户希望把外围控制块放在起始于地址 10000H 的存贮器空间中，则可以把重新定位寄存器编程为 1100H。在这样做了以后，就把集成的外围控制块中的寄存器，都移到了存贮器单元 10000H 到 100FFH。由于重新定位寄存器是处于外围控制块之中的，所以它也就随之被移到了存贮器空间的字单元 100FEH。

二、外围控制块寄存器

每个集成的外围部件的控制和状态寄存器，在外围控制块中，相对于外围控制块基地址，有一个固定的位置。在外围控制块中还有许多单元，没有分配给任何外围器件。若对这些单元中的任何一个进行写操作，则总线周期将照样进行。但值却不会存放到任何内部单元中去。这表示若紧接着对该单元进行读操作，该被写入的值，将不能被读出。

对访问集成的控制块中的一个单元的任何存贮器或 I/O 周期，处理器将执行一个外部总线周期。这表示有关的地址、数据和控制信息，将和执行一个“普通的”总线周期一样，在80186的外部引脚上被驱动。但是，由外部设备返回的任何信息将被忽略，即使是对一个不对应于任何集成的外围控制寄存器的单元进行访问也不例外。上述情况，对 80188 来说，除了在一个总线周期里所执行的对集成的寄存器作的字访问，是使用由写周期驱动的低 8 位数据(在外部总线中不存在高 8 位数据)以外都适用。

只要集成的外围器件被访问，处理器内部，就将产生一个准备就绪信号，因此，只要是对在集成的外围控制块中的单元进行访问，任何外部的准备就绪信号，就会被忽略。若对 256 字节外围控制块中不对应于任何集成的外围控制寄存器的单元进行访问，内部准备就绪信号也将被返回。除了对计时器寄存器进行访问以外，对于任何在集成的控制块中进行的访问，处理器都将插入 0 个等待状态。任何对计时器的控制、和计数寄存器的访问，都将产生一个等待状态。这个等待状态，是用来使处理器和计数器元件能正常地分路访问计时器控制寄存器的。

对集成的外围控制块所进行的任何访问，都必须是字访问。例如，不管操作码指定的是字

节写操作还是字写操作，对集成的寄存器的写操作，都将修改该寄存器的所有 16 位。从一个偶地址中读一个字节，不会产生问题，但是当从在外围控制块中的奇地址读一个字节时，所返回的数据就是没有定义的。这一点对 80186 和 80188 都适用。正如上面提到的那样，80188 尽管在外部是 8 位的数据总线，但内部它仍是一个 16 位的机器。因此，80188 执行的对集成的寄存器的字访问，也能在一个总线周期里完成，只不过仅在外部数据总线上，驱动低 8 位数据罢了。

§14.11.2 80186 同步信息

80186 的许多输入信号都是异步的，即不需要为了确保正常的功能而规定建立和保持时间。同每个这种输入相联系的是一个同步装置，这种同步装置取样外部的异步信号，并使它同步于 80186 的内部时钟。

一、为什么需要同步装置

为了能正常地操作，每个数据锁存器都需要一定的数据建立和保持时间，以便在规定的建立和保持时间的某一个窗口中，锁存器才真正对数据进行锁存。若输入在这个窗口中出现了变化，则在给定的输出延迟时间中，就不能获得稳定的输出。这个取样窗口的大小，一般比规定的要小得多，由于锁存器之间性能上的差别，所引起的窗口的移动，也将处于规定的窗口之中。

当一个数据锁存器正在试图锁存一个输入时，即使这个输入发生了变化，锁存器的输出在一定时间之后，也会达到稳定状态。一般来说，这个一定时间，比从选通到输出的延迟时间要长得多。图 14.78 表示了通常的输入到输出的选通变换，其中一个输入信号，在锁存器的取样窗口期间产生了一个变换。为了把一个异步信号同步，所需要做的是把这个信号，取样入一个数据锁存器，等待一定的时间，然后把它锁存入第二个数据锁存器。由于选通入第一个数据锁存器和选通入第二个数据锁存器之间的时间，使第一个数据锁存器能获得一个稳定的状态（或消除该异步信号），所以送达第二个数据锁存器的输入信号，就能满足第二个数据锁存器所需要的任何建立和保持时间。因此，第二个锁存器的输出就是相对于它的选通输入的一个同步信号。

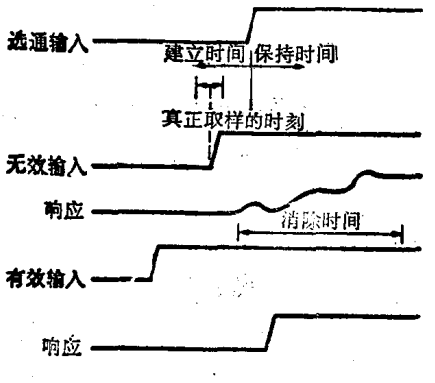


图 14 78 有效和无效的锁存器输入变换和响应

若在两个锁存器的选通信号之间，同步装置不能消除异步变换，同步就会失败。失败率由数据锁存器的取样窗口的真实大小、和两个锁存器的选通信号之间的时间决定。很明显，随着取样窗口变小，在取样窗口期间出现异步变换的次数就会下降。另外，较小的取样窗口也表示落入取样窗口的输入变换，有一个更快的消失时间。

二、80186 同步装置

80186 在 $\overline{\text{RES}}$、$\overline{\text{TEST}}$、TmrIn0-1、DRQ0-1、NMI、INT0-3、ARDY 和 HOLD 输入引脚上，接有同步装置，该装置使用上述的两级同步技术。锁存器的取样窗口规定为几十微微秒，这样

的规定，使同步装置在连续操作 30 年中，才有可能出现一次同步失败。

§14.11.3 80186 DMA 接口程序举例

```
$mod186
name                    assembly_example_80186_DMA_support
;
;       This file contains an example procedure which initializes the 80186 DMA controller
;to perform the DMA transfers between the 80186 system and the 8272 Floppy Disk Controller
;(FDC). It assumes that the 80186 peripheral control block has not been moved from its
;reset location.  这个文件含有一个作为例子的过程，该过程把 80186DMA 控制器初始化为
在 80186 系统和 8272 软磁盘控制器(FDC)之间进行 DMA 传送。它假定 80186 的外围控制
块没有被从它的复位位置移动过。
;
arg1                    equ     word ptr[BP+4]
arg2                    equ     word ptr[BP+6]
arg3                    equ     word ptr[BP+8]
DMA_FROM_LOWER          equ     0FFC0h          ; DMA register locations
                                                  DMA 寄存器单元
DMA_FROM_UPPER          equ     0FFC2h
DMA_TO_LOWER            equ     0FFC4h
DMA_TO_UPPER            equ     0FFC6h
DMA_COUNT               equ     0FFC8h
DMA_CONTROL             equ     0FFCAh
DMA_TO_DISK_CONTROL     equ     01486h          ; destination synchroniza-
                                                  tion 目同步
                                                ; source to memory, incre-
                                                  mented 源为存贮器，递增
                                                ; destination to I/O 目为
                                                  I/O
                                                ; no terminal count
                                                ; byte transfers 不传送终
                                                  止计数字节
DMA_FROM_DISK_CONTROL   equ     0A046h          ; source synchronization
                                                  源同步
                                                ; source to I/O 源为 I/O
                                                ; destination to memory, incr
                                                  目为存贮器，递增
                                                ; no terminal count 不传送
```

```
                                                    终止计数字节
                                                ; byte transfers
FDC__DMA        equ         6B8h                ; FDC DMA address FDC
                                                  DMA 地址
FDC__DATA       equ         688h                ; FDC data register FDC
                                                  数据寄存器
FDC__STATUS     equ         680h                ; FDC status register FDC
                                                  状态寄存器
cgroup          group       code
code            segment                         public 'code'
                public      set__dma__
                assume      cs:cgroup
;
;       set__dma (offset, to) programs the DMA channel to point one side to the disk DMA
;address, and the other to memory pointed to by ds: offset. If 'to' = 0 then will be a transfer
;from disk to memory if 'to' = 1 then will be a transfer from memory to disk. The parame-
;ters to the routine are passed on the stack. set__dma (offset, to) 编程 DMA 通道，一方面
指示磁盘 DMA 地址，另一方面由 ds:offset 指示存贮器。若 'to' = 0，则将是从磁盘到存贮
器的传送，若 'to' = 1，则将是从存贮器到磁盘的传送。送给这个程序的参数是在堆栈上传递
的。
;
set__dma__      proc        near
                enter       0,0                 ; set stack addressability 设
                                                  置可寻址的堆栈
                push        AX                  ; save registers used 保护被
                                                  使用的寄存器
                push        BX
                push        DX
                test        arg2,1              ; check to see direction of
                                                ; transfer
                                                  检查传送的方向
                jz          from__disk
; performing a transfer from memory to the disk controller
;
                mov         AX, DS              ; get the segment value 取
                                                  段值
                rol         AX, 4               ; gen the upper 4 bits of the
                                                ; physical address in the lo-
                                                ; wer 4
```

```
                                        ; bits of the register
                                          在寄存器的低4位取物理
                                          地址的高4位
        mov     BX, AX                  ; save the result...
                                          保护结果
        mov     DX, DMA__FROM__UPPER    ; prgm the upper 4 bits of
                                        ; the DMA source register
                                          编程DMA源寄存器的高
        out     DX, AX                    4位
        and     AX, 0FFF0h              ; form the lower 16 bits of
                                        ; the physical address
                                          形成物理地址的低16位
        add     AX, argl                ; add the offset  加上偏移量
        mov     DX, DMA__FROM__LOWER    ; prgm the lower 16 bits of
                                          the DMA source register
                                          编程DMA源寄存器的低
        out     DX, AX                    16位
        jnc     no__carry__from         ; check for carry out of addi-
                                        ; tion
                                          检查加法的进位
        inc     BX                      ; if carry out, then need to
                                          adj the upper 4 bits of the
                                          pointer
        mov     AX, BX                    若有进位,则需要调整指示
                                          器的高4位
        mov     DX, DMA__FROM__UPPER
        out     DX, AX
no__carry__from:
        mov     AX, FDC__DMA            ; prgm the low 16 bits of the
                                        ; DMA destination register
                                          编程DMA目寄存器的低
        mov     DX, DMA__TO__LOWER        16位
        out     DX, AX
        xor     AX, AX                  ; zero the up 4 bits of the
                                        ; DMA destination register
                                          把DMA目寄存器的高4
        mov     DX, DMA__TO__UPPER        位清0
        out     DX, AX
        mov     AX, DMA__TO__DISK__     ; prgm the DMA ctl reg  编
```

```
                    CONTROL                          程 DMA 控制寄存器
                mov DX, DMA__CONTROL                 ; note: DMA may begin
                                                     ; immediatly after this word
                                                     ; is output
                                                       注意：DMA 可在这个字被
                out DX, AX                             输出以后立即开始
                pop DX
                pop BX
                pop AX
                leave
                ret
from__disk:
;
; performing a transfer from the disk to memory  执行从磁盘到存贮器的一次传送
                mov AX, DS
                rol AX, 4
                mov DX, DMA__TO__UPPER
                out DX, AX
                mov BX, AX
                and AX, 0FFF0h
                add AX, arg1
                mov DX, DMA__TO__LOWER
                out DX, AX
                jnc no__carry__to
                inc BX
                mov AX, BX
                mov DX, DMA__TO__UPPER
                out DX, AX
no__carry__to
                mov AX, FDC__DMA
                mov DX, DMA__FROM__LOWER
                out DX, AX
                xor AX, AX
                mov DX, DMA__FROM__UPPER
                out DX, AX
                mov AX, DMA__FROM__DISK__
                    CONTROL
                mov DX, DMA__CONTROL
                out DX, AX
```

```
                pop     DX
                pop     BX
                pop     AX
                leave
                ret
set_dma_        endp
code            ends
                end
```

§14.11.4 80186计时器接口程序举例

```
$mod186
name            example_80186_timer_code
;
;       this file contains example 80186 timer routines The first routine sets up the timer and
;interrupt controller to cause the timer to generate an interrupt every 10 milliseconds, and
;to service interrupt to implement a real time clock. Timer 2 is used in this example because
;no input or output signals are required. The code example assumes that the peripheral con-
;trol block has not been moved from its reset location(FF00-FFFF in I/O space).这个文件含
有80186计时器程序的例子。第一个程序把计时器和中断控制器设置成每10毫秒使计时器
产生一次中断,并通过中断服务构成一个实时时钟。在这个例子中使用计时器2,因为不需要
输入输出信号。这个程序例子假定外围控制块没有被从它的复位位置(在I/O空间中的
FF00-FFFF)移动过。
arg1            equ     word ptr[BP+4]
arg2            equ     word ptr[BP+6]
arg3            equ     word ptr[BP+8]
timer_2int      equ     19              ; timer 2 has vector type 19
                                          计时器2具有向量类型19
timer_2control  equ     0FF66h
timer_2max_ctl  equ     0FF62h
timer_int_ctl   equ     0FF32h          ; interrupt controller regs
                                          中断控制器寄存器
eoi_register    equ     0FF22h
interrupt_stat  equ     0FF30h
data            segment                 public'data'
                public  hour_,minute_,second_,
                        msec_
msec_           db      ?
hour_           db      ?
```

```
minute__        db      ?
second__        db      ?
data            ends
cgroup          group   code
dgroup          group   data
code            segment                         public 'code'
                public  set__time__
                assume  cs:code, ds:dgroup
;
;       set__time (hour, minute, second) sets the time variables, initializes the 80186 timer 2
;to provide interrupts every 10 milliseconds, and programs the interrupt vector for timer 2
;set__time(hour, minute, secood)  设置时间变量,初始化 80186 计时器 2;以便每 10 毫秒提
供一次中断,并且为计时器 2 的中断向量编程.
set__time__     proc    near
                enter   0,0                     ; set stack addressability 设
                                                置可寻址的堆栈,
                push    AX                      ; save registers used 保护
                                                被使用的寄存器
                push    DX
                push    SI
                push    DS
                xor     AX, AX                  ; set the interrupt vector
                                                设置中断向量
                                                ; the timers have unique
                                                ; interrupt
                                                ; vectors even though they
                                                ; share
                                                ; the same control register
                                                即使计时器共享同一个控
                                                制寄存器,它们也只具有唯
                                                一的中断向量
                mov     DS, AX
                mov     SI, 4* timer2__int
                mov     DS: [SI], offset timer__2__
                        interrupt__routine
                inc     SI
                inc     SI
                mov     DS: [SI], CS
                pop     DS
```

```
                mov     AX, arg1                ; set the time values 设置
                                                  时间值
                mov     hour__, AL
                mov     AX, arg2
                mov     minute__,AL
                mov     AX, arg3
                mov     second__, AL
                mov     msec__,0
                mov     DX, timer2__max__ctl    ; set the max count value
                                                  设置最大计数值
                mov     AX, 20000               ; 10 ms/500 ns (timer 2 co-
                                                ; unts
                                                ; at1/4the CPU clock rate)
                                                  10 ms/500 ns (计时器 2 以
                                                  1/4 的 CPU 时钟速率计数)
                out     DX, AX
                mov     DX, timer2__control     ; set the control word 设置
                                                  控制字
                mov     AX, 1110000000000001b   ; enable counting 启动计数
                                                ; generate interrupts on TC
                                                ; continuous counting 在
                                                  TC 上产生中断继续计数
                out     DX, AX
                mov     DX, timer__int__ctl     ; set up the interrupt con-
                                                  troller 设置中断控制器
                mov     AX, 0000b               ; unmask interrupts 解除
                                                  中断
                                                ; highest priority interrupt
                                                  最高优先级中断
                out     DX, AX
                sti                             ; enable processor tnterrupts
                                                  允许处理器中断
                pop     SI
                pop     DX
                pop     AX
                leave
                ret
set__time__     endp
timer2__interrupt__ proc    far
```

```
routine
                push    AX
                push    DX
                cmp     msec__, 99          ; see if one second has passed
                                            看是否已过了一秒钟
                jae     bump__second        ; if above or equal… 若高
                                            于或等于…
                inc     msec__
                jmp     reset__int__ctl
bump__second:
                mov     msec__,0            ; reset mi'lisecond 复位毫
                                            秒
                cmp     second__, 59        ; see if one minute has passed
                                            看是否已过了一分钟
                jae     bump__minute
                inc     second__
                jmp     reset__int__ctl
bump__minute:
                mov     second__,0
                cmp     minute__, 59        ; see if one hour has passed
                                            看是否已过了一小时
                jae     bump__hour
                inc     minute__
                jmp     reset__int__ctl
bump__hour:
                mov     minute__,0
                cmp     hour__, 12          ; see if 12 hours have passed
                                            看是否已过了 12 小时
                jae     reset__hour
                inc     hour__
                jmp     reset__int__ctl
reset__hour:
                mov     hour__,1
reset__int__ctl:
                mov     DX, eoi__register
                mov     AX, 8000h           ; non-specific end of inter-
                                            rupt 中断的非专门结束
                out     DX, AX
                pop     DX
```

```
                    pop     AX
                    iret
timer2_interrupt_   endp
routine
code                ends
                    end
$mod186
name                example_80186_baud_code;
;
;       this file contains example 80186 timer routines. The second routine sets up the timer
;as a baud rate generator.In this mode, Timer 1 is used to continually output pulses with a
;period of 6.5 usec for use with a serial controller at 9600 baud programmed in divide by
;16 mode (the actual period required for 9600 baud is 6.51 usec). This assumes that the
;80186 is running at 8MHz. The code example also assumes that the peripheral control block
;has not been moved from its reset location(FF00—FFFF in I/O space).这个文件含有 80186
计时器程序的例子.第二个程序把计时器设置成一个波特率发生器.在这种方式下,计时器 1
被用来以 6.5 usec 的周期连续地输出脉冲,它与一个以模 16 除编程的 9600 波特的串行控制
器一起使用(9600 波特真正需要的周期是 6.51 秒). 这里假定 80186 是在 8MHz 下运行.这
个代码例子仍然假定外围控制块没有被从它的复位位置 (在 I/O 空间中的 FF00—FFFF) 移
动过.
timer 1_control     equ     0FF5Eh
timer 1_max_cnt     equ     0FF5Ah
code                segment                         public 'code'
                    assume  cs:code
; set_baud()initializes the 80186 timer 1 as a baud rate generator for
;       a serial port running at 9600 baud    Set_baud( )把 80186 的计时器 1 初始化
        成一个以 P600 波特运行的专用转接口的波特率发生器
set_baud_           proc    near
                    push    AX                  ; save registers used 保护被
                                                  使用的寄存器
                    push    DX
                    mov     DX, timer 1_max_cnt ; set the max count value
                                                  设置最大计数值
                    mov     AX, 13              ; 500 ns*13 = 6.5 usec
                    out     DX, AX
                    mov     DX, timer 1_control ; set the control word 设置
```

```
                                                    控制字
                mov     AX, 1100000000000001b       ; enable counting 启动计数
                                                    ; no interrupt on TC 在TC
                                                      上无中断
                                                    ; continuous counting 继续
                                                      计数
                                                    ; single max count register
                                                      单个最大计数寄存器
                out     DX, AX
                pop     DX
                pop     AX
                ret
set_baud_       endp
code            ends
                end
$mod186
name            example_80186_count_code
;
;       this file contains example 80186 timer routines. The third routine sets up the timer as
an external event counter. In this mode, Timer 1 is used to count transitions on its input
pin. After the timer has been set up by the routine, the number of events counted can be
directly read from the timer count register at location FF58H in I/O space. The timer will
count a maximum of 65535 timer events before wrapping around to zero. This code example
also assumes that the peripheral control block has not been moved from its reset location
(FF00—FFFF in I/O space). 这个文件含有80186计时器程序的例子,这第三个程序把计时
器设置为一个外部事件计数器. 在这种方式下,计时器1被用来记数在它的输入引脚上的变
换. 在计时器被程序设置以后,已被计数的事件数可以从在I/O空间中的FF58H单元的计
时器计数寄存器中直接读出. 在计时器计数绕回到0以前,计时器能计数的最大计时器事件
数是65535. 这个代码例子仍假定外围控制块没有被从它的复位位置(在I/O空间中的FF00
—FFFF)移动过.
;
timer 1_control     equ     0FF5Eh
timer 1_max_cnt     equ     0FF5Ah
timer 1_cnt_reg     equ     0FF58H
code                segment                         public 'code'
                    assume  cs:code
; set_count( )initializes the 80186 timer 1 as an event counter    Set_count( )把80186
  的计时器1初始化成一个事件计数器
```

```
set__count__    proc    near
                push    AX                      ; save registers used 保护
                                                  被使用的寄存器
                push    DX
                mov     DX, timer 1__max__cnt   ; set the max count value
                                                  设置最大计数值
                mov     AX, 0                   ; allows the timer to coune
                                                ; all the way to FFFFH
                                                  使计时器能计数达FFFFH
                out     DX, AX
                mov     DX, timer 1__control    ; set the control word 设置
                                                  控制字
                mov     AX, 1100000000000101b   ; enable counting 启动计数
                                                ; no interrupt on TC 在TC
                                                  上无中断继续计数
                                                ; continuous countiug
                                                ; single max count register
                                                  单个最大计数寄存器
                                                ; external clocking 外部定
                                                  时
                out     DX, AX
                xor     AX, AX                  ; zero AX AX清0
                mov     DX, timer 1__cnt__reg   ; and zero the count in the
                                                ; timer count register
                                                  并把在计时器计数寄存器
                out     DX, AX                    中的计数清0
                pop     DX
                pop     AX
                ret
set__count__    endp
code            ends
                end
```

§14.11.5 80186中断控制器接口程序举例

```
$mod186
name            example__80186__interrupt__code
;
;       This routine configures the 80186 interrupt controller to provide two cascaded inter-
```

```
;rupt inputs (through an external 8259A interrupt controller on pins INT0/INT2) and two
;direct interrupt inputs(on pins INT1 and INT3).The default priority levels are used. Because
;of this, the priority level programmed into the control register is set the 111, the level all
;interrupts are programmed to at reset. 这个程序把80186中断控制器配置成提供两个级联
中断输入(通过一个在INT0/INT2引脚上的外部8259A中断控制器)和两个直接中断输入
(在INT1和INT3引脚上)。使用缺省优先级。因此，编程入控制寄存器的优先级被设置为
111,也就是在复位时所有的中断被编程的优先级。
;
int0_control        equ     0FF38H
int_mask            equ     0FF28H
;
code                segment                         public 'code'
                    assume  CS:code
set_int_            proc    near
                    push    DX
                    push    AX
                    mov     AX, 0100111B            ; cascade mode 级联方式
                                                    ; interrupt unmasked 解除
                                                      中断屏蔽
                    mov     DX, int0_control
                    out     DX, AX
                    mov     AX, 01001101B           ; now unmask the other ex-
                                                    ; ternal interrupts
                                                      现在解除其他外部中断的
                                                      屏蔽
                    mov     DX, int_mask
                    out     DX,AX
                    pop     AX
                    pop     DX
                    ret
set_int_            endp
code                ends
                    end
$mod186
name                example_80186-interrupt_code
;
;       This routine configures the 80186 interrupt controller into iRMX86 mode. This code
;does not initialize any of the 80186 integrated peripheral control registers, nor does it initia-
;lize the external 8259A or 80130 interrupt controller. 这个程序把80186中断控制器配置
```

成 iRMX86 方式。这个程序既不初始化任何 80186 集成的外围控制寄存器，也不初始化外部的 8259A 或 80130 中断控制器。

```
;
relocation_reg      equ     0FFFEH
;
code                segment                         public 'code'
                    assume  CS:code
set_rmx_            proc    near
                    push    DX
                    push    AX
                    mov     DX, relocation_reg
                    in      AX, DX                  ; read old contents of register 读寄存器原来的内容
                    or      AX, 0100000000000000B   ; set the RMX mode bit 设置 RMX 方式位
                    out     DX, AX
                    pop     AX
                    pop     DX
                    ret
set_rmx_            endp
code                ends
                    end
```

§14.11.6 80186/8086 系统初始化程序举例

```
name                example_80186_system_init
;
;       This file contains a system initialization routine for the 80186 or the 8086. The code
;determines whether it is running on an 80186 or an 8086, and if it is running on an 80186,
;it initializes the integrated chip select registers. 这个文件含有一个 80186 或 8086 的系统初始化程序。由程序决定它是在 80186 上还是在 8086 上运行，若它在 80186 上运行，它将初始化集成的片选寄存器。
;
restart             segment at                      0FFFFh
;
; This is the processor reset address at 0FFFF0H
;
                    org     0
                    jmp     far ptr initialize
```

```
restart         ends
;
                extrn   monitor:far
init_hw         segment at                      0FFF0h
                assume  CS:init_hw
;
;       This segment initializes the chip selects. It must be located in the top 1K to insure that
;the ROM remains selected in the 80186 system until the proper size of the select area can be
;programmed. 这段初始化片选，它必须被放在最高的 1K 以确保在 80186 系统中保持选中
该 ROM 直到选中区域的适当容量可以被编程为止。
;
UMCS_reg        equ     0FFA0H          ; chip select register locations
                                        片选寄存器单元
LMCS_reg        equ     0FFA 2 H
PACS_reg        equ     0 FFA 4 H
MPCS_reg        equ     0 FFA 8 H
UMCS_value      equ     0 F 800 H       ; 64K, no wait states 64K,
                                        无等待状态
LMCS_value      equ     07 F 8 H        ; 32 K, no wait states 32 K
                                        无等待状态
PACS_value      equ     72 H            ; peripheral base at 400H,
                                        外围基地址在 400 H
MPCS_value      equ     0BAH            ; PCS5 and 6 supplies,
                                        ; peripherals in I/O space 支
                                        持 PCS 5 和 6，外围在 I/O
                                        空间
initialize      proc    far
                mov     AX, 2           ; determine if this is an
                mov     CL, 33          ; 8086 or an 80186 (checks
                shr     AX, CL          ; to see if the multiple bit
                test    AX, 1           ; shift value was ANDed)
                                        判断是 8086 还是 80186
                                        (检查多位移位值是否被
                                        "与"过)
                jz      not_80186
                mov     DX, UMCS_reg    ; program the UMCS regis-
                                        ter 编程 UMCS 寄存器
                mov     AX, UMCS_value
```

```
                out     DX, AX
                mov     DX, LMCS__reg          ; program the LMCS register
                                               ; 编程 LMCS 寄存器
                mov     AX, LMCS__value
                out     DX, AX
                mov     DX, PACS-reg           ; set up the peripheral chip
                                               ; selects (note the mid-range
                                               ; memory chip selects are not
                                               ; needed in this system, and
                                               ; are thus not initialized) 设
                                               ; 置外围片选信号(注意,在
                                               ; 这个系统中中部片选信号
                                               ; 不需要,因而没有被初始
                                               ; 化)
                mov     AX, PACS-value
                out     DX, AX
                mov     DX, MPCS-reg
                mov     AX, MPCS__value
                out     DX, AX
;
;       Now that the chip selects are all set up, the main program of the computer may be
 executed. 现在片选信号都已设置,计算机的主程序可以执行了。
;
not__80186:
                jmp     far ptr monitor
initialize      endp
init__hw        ends
                end
```

§14.11.7 80186 等待状态特性

由于 80186 含有分离的总线接口和执行部件,所以处理器的性能,不会随着处理器的存贮器周期中加入等待状态,而以恒定的比率下降。性能下降的真实比率,将依赖于所遇到的用户程序中的指令的类型和混合程度。

下面所示的是两个 80186 的汇编语言程序和当等待状态被加入到处理器的存贮器周期中时,该两程序的真实执行时间。这些程序显示了引进等待状态后,影响或不影响系统性能的两种极端情况。

程序 1,集中在存贮器操作上。它执行了许多广泛使用各种处理器寻址方式的存贮器读和存贮器写指令。结果,执行部件就必须常常等待总线接口部件去取指令和执行存贮器周期,

以便能继续运行。因此，这种类型的程序执行时间，会随着等待状态的加入而迅速增长，因为执行时间几乎全部多到处理器执行总线周期的速度的限制。

相反，程序 2。是集中在 CPU 操作上的。它执行了许多整数乘法，在执行整数乘法期间，总线接口部件，就能和执行部件并行地操作，从而把指令预取队列装满。在这个程序里，总线接口部件执行总线操作的速度，比执行部件对它的要求更快。出现这种情况时，等待状态加到存贮器接口对处理器性能下降所产生的影响就要小得多。这个程序的执行时间，接近于把每条指令所需要的时钟周期数相加的总和，因为执行部件不需要经常等待总线接口部件把一个操作码放到预取队列。这样，执行部件为等待一条指令，而浪费的时钟周期就大大减少。表 14.7 列出了以 8MHz 运行的 80186 在引入等待周期以后，测量到的这两个程序的执行时间。

表 14.7 两个程序执行时间的比较

等待状态数	程序 1		程序 2	
	执行时间(μsec)	性能下降	执行时间(μsec)	性能下降
0	505		294	
1	595	18%	311	6%
2	669	12%	337	8%
3	752	12%	347	3%

```
$mod186
name                    example_wait_state_performance
;
;       This file contains two programs which demonstrate the 80186 performance degradation
;as wait states are inserted. Program 1 performs a transformation between two types of cha-
;racters sets, then copies the transformed characters back to the original buffer (which is 64
;bytes long).Program 2 performs the same type of transformation, however instead of perfor-
;ming a table lookup. it multiplies each number in the original 32 word buffer by a constant
;(3,note the use of the integer immediate multiply instruction). Program"nothing"is used to
;measure the call and return times from the driver program only. 这个文件含有两个程序，
它们演示了随着等待状态的插入 80186 性能的降低。程序 1 实现在两种类型的字符集之间的
变换，然后把变换过的字符拷贝回原来的缓冲器（64 字节长）。程序 2 实现相同类型的变换，
但是它把在原来 32 个字缓冲器中的每个数乘以一个常数（3，注意整数立即数乘法指令的用
法）而不是查表。程序 "nothing" 只用来计算从驱动程序中调用和返回的时间。
;
cgroup          group   code
dgroup          group   data
data            segment                         public 'data'
t_table         db      256 dup(?)
t_string        db      64 dup(?)
m_array         dw      32 dup(?)
```

```
data            ends
code            segment                         public'code'
                assume  CS:cgroup, DS:dgroup
                public  bench__1, bench__2, nothing__,
                        wait__state__, set__timer__
bench__1        proc    near
                push    SI                      ; save registers used  保护被
                                                  使用的寄存器
                push    CX
                push    BX
                push    AX
                mov     CX, 64                  ; translate 64 bytes  翻译64
                                                  个字节
                mov     SI, 0
                mov     BH, 0
loop__back:
                mov     BL, t__string[SI]       ; get the byte  取字节
                mov     AL, t__table[BX]        ; translate byte  翻译字节
                mov     t__string[SI], AL       ; and store it  并存放
                inc     SI                      ; increment index
                loop    loop__back              ; do the next byte  递增变
                                                  址并对下一个字节执行
                pop     AX
                pop     BX
                pop     CX
                pop     SI
                ret
bench__1        endp
bench__2        proc    near
                push    AX                      ; save registers used  保护
                                                  被使用的寄存器
                push    SI
                push    CX
                mov     CX, 32                  ; multiply 32 numbers  对
                                                  32个数做乘法
                mov     SI, offset m__array
loop__back__2:
                imul    AX, word ptr [SI], 3    ; immediate multiply  立即
                                                  数乘法
```

```
                mov     word ptr [SI], AX
                inc     SI
                inc     SI
                loop    loop_back_2
                pop     CX
                pop     SI
                pop     AX
                ret
bench_2_        endp
nothing_        proc    near
                ret
nothing_        endp
;
;       wait_state (n) sets the 80186 LMCS register to the number of wait states (0 to 3)
;indicated by the parameter n (which is passed on the stack). No other bits of the LMCS
;register are modified. wait_state (n) 把 80186 LMCS 寄存器设置为由参数 n (它在堆栈上
 传递)指定的等待状态数(0 到 3)。LMCS 寄存器的其他位不被修改
;
wait_state_     proc    near
                enter   0,0             ; set up stack frame 建立堆
                                          栈框架
                push    AX              ; save registers used 保护被
                                          使用的寄存器
                push    BX
                push    DX
                mov     BX. word ptr [BP+4]     ; get argument 取变量
                mov     DX, 0FFA2h      ; get current LMCS register
                                          取当前的 LMCS 寄存器
contents
                in      AX, DX
                and     AX, 0FFFCh      ; and off existing ready bits
                                          "与"去存在的准备就绪位
                and     BX, 3           ; insure ws count is good 确
                                          保字计数是好的
                or      AX, BX          ; adjust the ready bits 调整
                                          准备就绪位
                out     DX, AX          ; and write to LMCS 并写
                                          LMCS
                pop     DX
```

```
                pop     BX
                pop     AX
                leave                           ; tear down stack frame 放
                                                  弃堆栈框架
                ret
wait_state_     endp
;
;       set_timer() initializes the 80186 timers to count microseconds. Timer 2 is set up as a
;prescaler to timer 0, the microsecond count can be read directly out of the timer 0 count re-
;gister at location FF50H in I/O space. set_timer() 初始化 80186 计时器以计数微秒. 计时
器 2 被设置为计时器 0 的前置计时器,微秒计数可以直接从在 I/O 空间中的 FF50H 单元中
读出.
;
set_timer_      proc    near
                push    AX
                push    DX
                mov     DX, 0ff66h              ; stop timer 2  停止计时器 2
                mov     AX, 4000h
                out     DX, AX
                mov     DX, 0ff50h              ; clear timer 0 count  清除
                                                  计时器 0 计数
                mov     AX, 0
                out     DX, AX
                mov     DX, 0ff52h              ; timer 0 counts up to 65535
                                                  计时器 0 计数到 65535
                mov     AX, 0
                out     DX, AX
                mov     DX, 0ff56h              ; enable timer 0  启动计时
                                                  器 0
                mov     AX, 0c009h
                out     DX, AX
                mov     DX, 0ff60h              ; clear timer 2 count  清除
                                                  计时器 2 计数
                mov     AX, 0
                out     DX, AX
                mov     DX, 0ff62h              ; set maximum count of
                                                  timer 2  设置计时器 2 的
                                                  最大计数
                mov     AX, 2
```

```
                out     DX, AX
                mov     DX, 0ff66h          ; re_enable timer 2  重新启
                                              动计时器 2
                mov     AX, 0c001h
                out     DX, AX
                pop     DX
                pop     AX
                ret
set_timer_      endp
code            ends
                end
```

§14.11.8 80186的新指令

80186在8086指令集的基础上，新加了一些指令，在这里我们对这些新的指令的操作作一些解释。为了使8086/186汇编语言，能使用这些新的指令，必须在汇编语言文件的开头，放上一行"$mod186"。

1. PUSH 立即数

这条指令使立即数据能被推入处理器堆栈。立即数据可以是立即字节、或立即字数据。若该数据是一个字节，则它在被推入堆栈以前，将被符号扩展（因为所有的堆栈操作都是字操作）。

2. PUSHA,POPA

这些指令能使80186把所有的通用寄存器保护到堆栈中，或从堆栈中恢复。由指令PUSHA保护的寄存器（以它们被推入堆栈的次序），是AX、CX、DX、BX、SP、BP、SI和DI。推入堆栈的SP的值是第一个PUSH(AX)以前的值。弹出的SP寄存器的值将被忽略。

PUSHA指令并不保护任何段寄存器（CS、DS、SS、ES）、指令指示器(IP)、标志寄存器，或任何集成的外围寄存器。

3. IMUL 立即数

这条指令使一个值能乘以一个立即数，这种操作的结果是16位的。这条指令的一个操作数，是用80186的某一种寻址方式获得的（表示它可以在一个寄存器中、或在存贮器中）。这个立即数值，可以是一个字节、或一个字，但若是一个字节，则要进行符号扩展。这种乘法的结果，可以放在任何80186的通用、或指示寄存器中。

这条指令需要三个操作数：放结果的寄存器，立即数值和第二个操作数。第二个操作数也可以是任何80186通用寄存器、或指定的存贮器单元。

4. 由一个立即数值决定的移位/环移指令

80186可以执行移位位数由一个立即数值指定的多位移位、或循环移位，这是和8086不同的。在8086中，只能执行一位的移位、或移位位数由CL寄存器指定的多位移位或环移。

所有80186的移位/环移指令，都可以由一个立即数值来指定移位的位数。和80186所执行的所有多位移位一样，移位的位数，是模32所指定的位数（即，80186多位移位的最大移位

数是31)。这些指令需要两个操作数：被移位的操作数（可以是寄存器或存贮器单元）和要移位的位数。

5．块输入/输出

80186加了两条新的输入/输出指令：INS和OUTS。这些指令执行块输入或输出操作。它们的操作和处理器的串传送操作相似。

INS指令，执行从一个I/O转接口到存贮器的块输入。I/O地址由DX寄贮器指定，而存贮器单元则由DI寄存器指示。在操作执行以后，DI寄存器被调整1(若输入的是一个字节)、或2（若输入的是一个字）。这种调整由处理器标志寄存器中的方向位决定是递增还是递减。ES寄存器用来寻址存贮器，并且是不能被超越的。当前面放有REP前缀时，这条指令能把数据块，从一个I/O地址传送到一个存贮器块。要注意的是，这种操作不修改在DX寄存器中的I/O地址。

OUTS指令，执行从存贮器到一个输出转接口的块输出。I/O地址由DX寄存器指定，存贮器单元由SI寄存器指定。在执行操作以后，SI寄存器被调整1（若输出的是一个字节)、或2（若输出的是一个字）。这种调整，由处理器标志寄存器中的方向位决定是递增还是递减。DS段寄存器被用来寻址存贮器，但不能使用一个段超越前缀来超越。当在前面放有REP前缀时，这条指令能把在一个存贮器块中的数据块，传送到一个I/O地址。同样，这种操作也不修改在DX寄存器中的I/O地址。

象串传送指令一样，不管是进行字还是字节操作，这两条指令都需要两个操作数。此外，决定是字节还是字操作可以在基本助记符后面加上“B”或“W”来直接用助记符表示，例如：

```
INSB            ；执行字节输入
REP OUTSW       ；执行字块输出
```

6．BOUND

80186提供了一条BOUND指令，使数组的边界检查更加方便。使用这条指令时，由计算得到的进入一个数组的索引，被放在80186的一个通用寄存器中，而该数组的索引的上界和下界，则放在存贮器的相连的两个字单元中。BOUND指令把寄存器的内容和存贮器单元中的内容进行比较，若寄存器中的值，并不介于存贮器单元中的两个值之间，就会发生中断类型为5的中断。所执行的比较是SIGNED比较。等于上界或下界的寄存器值将不会引起一个中断。

这条指令需要两个变量：存放计算得到的数组索引的寄存器、和含有数组下界的存贮器字单元（它可以由任何80186存贮器寻址方式指定）。含有数组上界的存贮器单元，必须紧接在含有数组下界的存贮器单元后面。

7．ENTER和LEAVE

80186含有两条用来建立和放弃高级、块结构语言的堆栈框架的指令。用来建立这种堆栈框架的指令是ENTER。关于这条指令的算法是：

```
PUSH    BP      /*  保护先前框架的指示器*/
if  level = 0   then
    BP: = SP;
else    temp1: = SP;        /*保护当前框架的指示器*/
```

```
temp2: = level - 1;
do while temp2 > 0 /*拷贝先前层的框架*/
BP: = BP - 2;      /*指示器*/
PUSH[BP];
BP: = temp1;
PUSH BP;           /*放好当前层的指示器*/
                   /*在保护区域*/
SP: = SP - disp;   /*为局部变量产生堆栈空间*/
```

图 14.79 表示了堆栈在这个操作前后的结构.

这条指令需要两个操作数：第一个值 (disp)，指定这个程序的局部变量所需要的字节数。这是一个无符号数，并且最大能为 65535.第二个值(层)，也是一个无符号值，它指定着过程的层次，并且最大可以为 255.

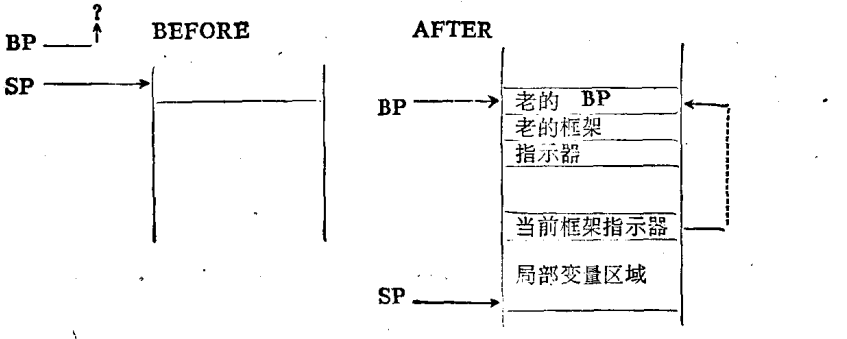


图 14.79 ENIER 指令堆栈框架

80186 的 LEAVE 指令，用来放弃由 INTER 指令建立的堆栈框架。从执行 ENTER 指令以后的堆栈结构可见，执行这条指令，只要把 BP 寄存器的内容传送到 SP 寄存器，并从堆栈中弹出老的 BP 值就可以了.

ENTER 指令和 LEAVE 指令，都不保护 80186 的通用寄存器。若它们必须被保护，则要用另外的指令来实现这种操作。此外，LEAVE 指令也不执行从一个子程序中返回的功能。若希望有这种功能的话，则 LEAVE 指令之后必须直接跟一条 RET 指令.

§14.11.9 80186/80188 的区别

80188 除了它只有 8 位的外部总线以外，基本上同 80186 相同。它具有和 80186 相同的执行部件、计时器、外围控制块、中断控制器、片选和 DMA 逻辑。由较窄的数据总线造成的 80188 和 80186 的区别是：

1. 80188 只有一个 4 字节的预取队列，而 80186 具有一个 6 字节的预取队列。80188 预取队列只有 4 个字节的原因，是 80188 在一个时刻只预取 1 个字节的操作码，所以用来填满 80188 的较小队列所需要的总线周期，反而比用来填满 80186 的队列的总线周期多。结果，为了防止在一个跳转期间被预取操作码浪费过多的总线周期，就只好使用一个较小的队列.

2. 80186 上的 AD_8—AD_{15}，在 80188 上变成了 A_8—A_{15}。在整个 80188 的总线周期中，这些引脚上始终存在着有效的地址信息，但在空闲 T 周期期间，不能保证地址信息有效.

3．$\overline{BHE}$/S7 在 80188 中总是高电平，因为数据总线的高 8 位是不存在的。

4．80188 的 DMA 控制器，只执行字节传送。在 DMA 控制字中的 $\overline{B}$/W 位是被忽略的。

5．许多存贮器访问指令的执行时间，在 80188 中要比在 80186 中长，因为存贮器访问，必须通过较窄的数据总线进行。80188 也将比 80186 更多地受到总线的限制（即执行部件将需要更多地等待从存贮器中取出操作码），因为数据总线只有 8 位。但是在处理器内部的执行时间，80188 和 80186 是相同的。

必须指出的是：80188 在内部是一个 16 位处理器，这就是说，对 80188 集成的外围寄存器的访问将是按 16 位的，而不是按 8 位进行的。所有内部的外围寄存器仍然是 16 位宽，并且只要一个读或写周期，就能访问这些寄存器。因此，对这些内部寄存器进行访问时，只要运行一个总线周期，并且只有写数据的低 8 位将被在外部总线上驱动。对在集成的外围控制块中的寄存器的访问，都必须是字访问。

§14.11.10 80186 特 性

1．直流特性

表 14.8 80186 直流特性（$T_A=0°-70°C, V_{CC}=5V\pm10\%$）

符 号	参 数	最 小	最 大	单 位	测 试 条 件
V_{IL}	Input Low Voltage	−0.5	+0.8	Volts	
V_{IH}	Input High Voltage (All except X1 and $\overline{RES}$)	2.0	V_{CC}+0.5	Volts	
V_{IH1}	Input High Voltage($\overline{RES}$)	TBD	V_{CC}+0.5	Volts	
V_{OL}	Output Low Voltgue		0.45	Volts	I_{OL}=2.5 mA for $\overline{S0}$-$\overline{S2}$ I_{OL}=2.0 mA for all other outputs
V_{OH}	Output High Voltage		2.4	Volts	I_{OH}=−400μA
I_{CC}	Power Supply Current		550	mA	T_A=25℃
I_{LI}	Input Leakage Current		±10	μA	$0V<V_{IN}<V_{CC}$
I_{LO}	Output Leakage Current		±10	μA	$0.45V<V_{OUT}<V_{CC}$
V_{CLO}	Clock Output Low		0.6	Volts	I_{OL}=2.5mA
V_{CHO}	Clock Output Hcigh	4.0		Volts	I_{OH}=−200μA
V_{CL1}	Clock Input Low Voltage	−0.5	0.6	Volts	
V_{CH1}	Clock Input High Voltage	3.9	V_{CC}+1.0	Volts	
C_{IN}	Input Capacitance		10	pF	
C_{IO}	I/O Capacitance		20	pF	

2. 交流特性 ($T_A=0°-70°, V_{CC}=5V\pm10\%$)

表 14.9 80186 定时要求

符 号	参 数	最 小	最 大	单 位	测试条件
TDVCL	Data in Setup(A/D)	20		ns	
TCLDX	Data in Hold(A/D)	10		ns	
TARYHCH	Asynchronous Ready(AREADY) active setup time*	20		ns	
TARYLCL	AREADY inactive setup time	35		ns	
TCHARYX	AREADY hold time	15		ns	
TSRYCL	Synchronous Ready (SREADY)transition setup time	35		ns	
TCLSRY	SREADY transition hold time	15		ns	
THVCL	HOLD Setup*	25		ns	
TINVCH	INTR, NMI, $\overline{TEST}$, TIMERIN, Setup*	25		ns	
TINVCL	DRQ0, DRQ1, Setup*	25		ns	

* 为了保证在下个时钟识别

表 14.10 80186 主设备接口定时响应

符 号	参 数	最 小	最 大	单 位	测 试 条 件
TCLAV	Address Valid Delay	10	44	ns	C_L=20-200pF all outputs
TCLAX	Address Hold	10		ns	
TCLAZ	Address Float Delay	TCLAX	35	ns	
TCHCZ	Command Lines Float Delay		45	ns	
TCHCV	Command Lines Valid Delay (after float)		55	ns	
TLHLL	ALE Width	TCLCL-35		ns	
TCHLH	ALE Active Delay		35	ns	
TCHLL	ALE Inactive Delay		35	ns	
TLLAX	Address Hold to ALE Inactive	TCHCL-25		ns	
TCLDV	Data Valid Delay	10	44	ns	
TCLDOX	Data Hold Time	10		ns	
TWHDX	Data Hold after $\overline{WR}$	TCLCL-40		ns	
TCVCTV	Control Active Delay1	10	70	ns	
TCHCTV	Control Active Delay2	10	55	ns	
TCVCTX	Control Inactive Delay	10	55	ns	
TAZRL	Address Float to $\overline{RD}$ Active	0		ns	
TCLRL	$\overline{RD}$ Active Delay	10	70	ns	

（续表）

符　号	参　数	最　小	最　大	单　位	测　试　条　件
TCLRH	$\overline{RD}$ Inactive Delay	10	55	ns	
TRHAV	$\overline{RD}$ Inactive to Address Active	TCLCL-40		ns	
TCLHAV	HLDA Valid Delay	10	50	ns	
TRLRH	$\overline{RD}$ Width	2TCLCL-50		ns	
TWLWH	$\overline{WR}$ Width	2TCLCL-40		ns	
TAVAL	Address Valid to ALE Low	TCLCH-25		ns	
TCHSV	Status Active Delay	10	55	ns	
TCLSH	Status Inactive Delay	10	55	ns	
TCLTMV	Timer Output Delay		60	ns	100pf max
TCLRO	Reset Delay		60	ns	
TCHQSV	Queue Status Delay		35	ns	

表 14.11　80186 片选定时响应

符　号	参　数	最　小	最　大	单　位	测　试　条　件
TCLCSV	Chip-Select Active Delay		66	ns	
TCXCSX	Chip-Select Hold from Command Inactive	35		ns	
TCHCSX	Chip-Select Inactive Delay	10	35	ns	

表 14.12　80186 CLKIN 要求

符　号	参　数	最　小	最　大	单　位	测　试　条　件
TCKIN	CLKIN Period	62.5	250	ns	
TCKHL	CLKIN Fall Time		10	ns	3.5 to 1.0 volts
TCKLH	CLKIN Rise Time		10	ns	1.0 to 3.5 volts
TCLCK	CLKIN Low Time	25		ns	1.5volts
TCHCK	CLKIN High Time	25		ns	1.5volts

表 14.13　80186 CLKOUT 定时(200pF 负载)

符　号	参　数	最　小	最　大	单　位	测　试　条　件
TCICO	CLKIN to CLKOUT Skew		50	ns	
TCLCL	CLKOUT Period	125	500	ns	
TCLCH	CLKOUT Low Time	$^1/_2$TCLCL-7.5		ns	1.5 volts
TCHCL	CLKOUT High Time	$^1/_2$TCLCL-7.5		ns	1.5 volts
TCH1CH2	CLKOUT Rise Time		15	ns	1.0 to 3.5 volts
TCL2CL1	CLKOUT Fall Time		15	ns	3.5 to 1.0 volts

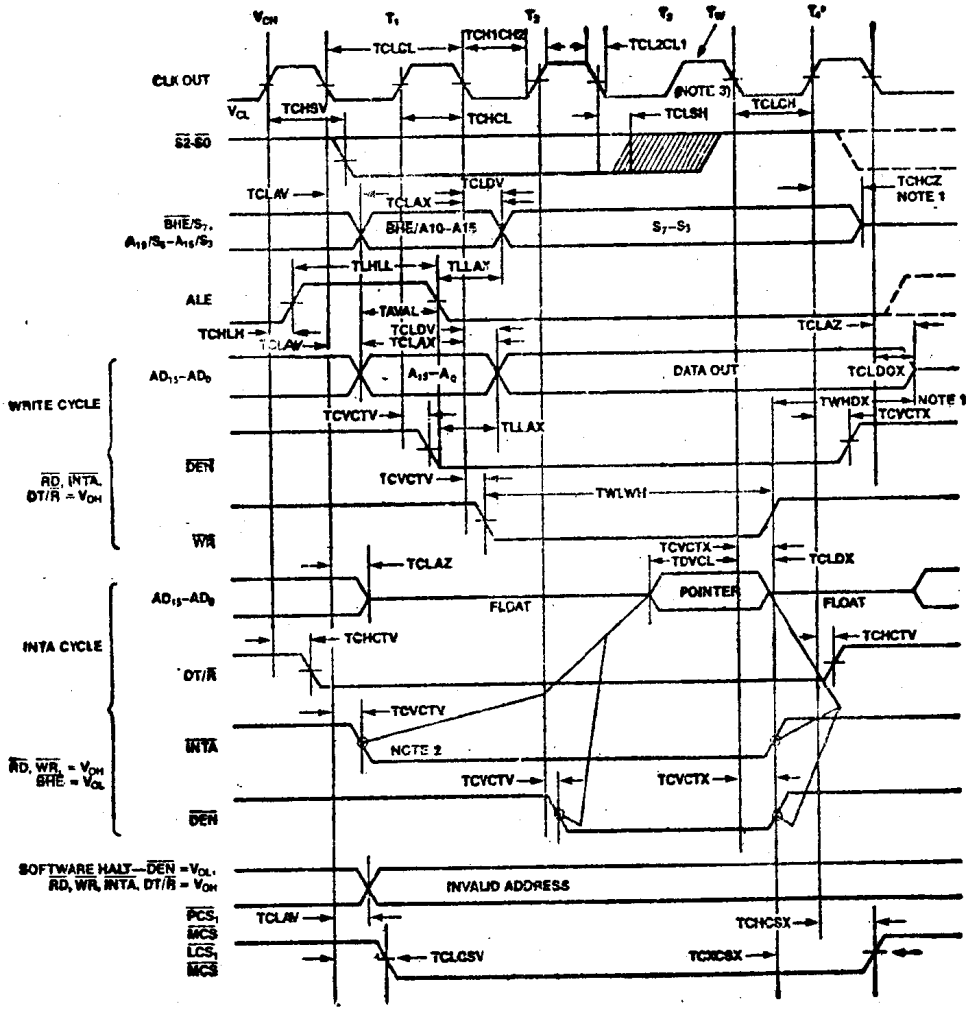
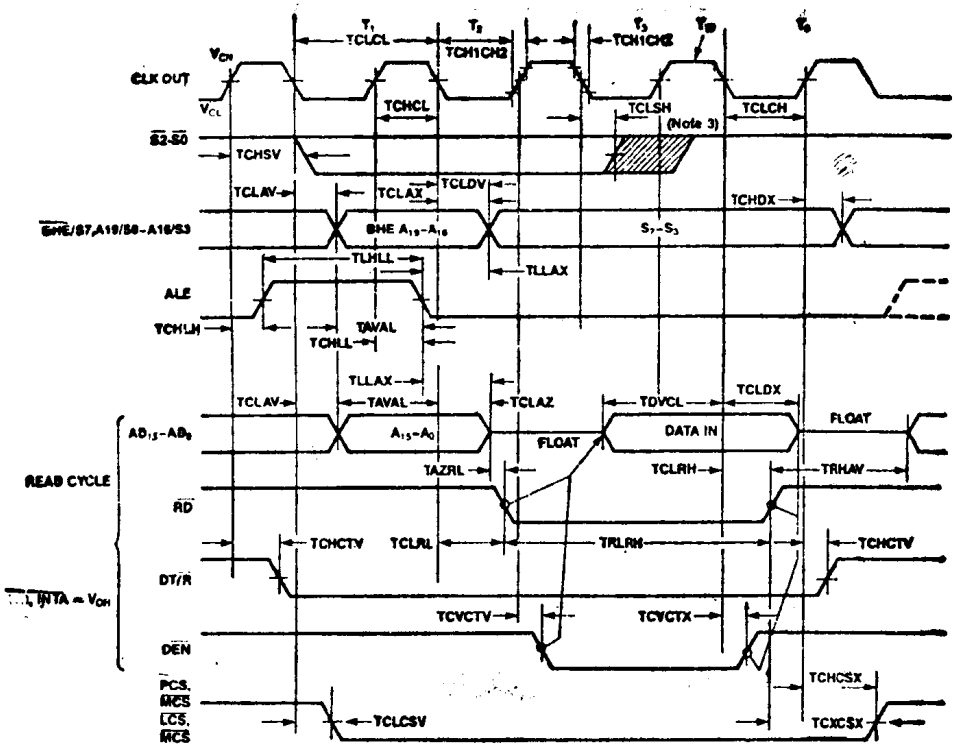


图 14.80A　80186 主周期定时



注：1. 在写周期之后，仅当 80186 进入一个"保持响应"状态之后，80186 才浮空局部总线

2. 在 RMX 方式下，INTA 晚一个时钟出现

3. T_4 前状态不活跃

图 14.80 B　80186 主周期定时

思　考　题

1. 80186 芯片集成了哪些部件？它们的作用各是什么？
2. 80186 和 80188 的数据总线操作，有些什么不同？
3. 为什么在 80186 系统中，没有分别为存贮器和 I/O 提供 $\overline{RD}$、$\overline{WR}$ 指令？若要分别为存贮器和 I/O 提供 $\overline{RD}$ 和 $\overline{WR}$ 指令，可以用什么方法来实现？
4. 80186 产生的 ALE 信号同 80186 产生的 ALE 信号相比，有些什么不同？
5. 80186 产生的 $\overline{LOCK}$ 信号和 8086 产生的 $\overline{LOCK}$ 信号，在实现总线封锁时，有些什么不同？
6. 什么是总线等待时间？什么是中断等待时间？
7. 80186 的 DMA 部件有几个通道？每个通道含有些什么寄存器，它们的作用各是什么？
8. DMA 传送的简单过程是怎样的？
9. 举例说明什么是目同步的 DMA 传送，什么是源同步的 DMA 传送？
10. 80186 的计时器部件，由哪些计时器组成，它们各能承担什么任务，试举例说明。
11. 每个计时器含有哪些寄存器，它们的作用各是什么？
12. 80186 集成的中断控制器，有几种主要的工作方式？它们工作的主要特点各是什么？
13. 中断控制器中有哪些寄存器，它们的作用各是什么？
14. 80186 中断控制器的四个外部中断引脚，在直接输入方式、级联方式、SFNM 方式和 iRMX86 方式中各起什么作用？

15. 80186片选部件，提供哪些片选信号，它们的用途各是什么？

16. 8086和80186在执行指令时有哪些区别？

17. 80186的外围控制块的作用是什么？对这些寄存器的寻址，有些什么特点？它是用什么机构来实现在存贮器和I/O空间浮动的？

18. 与8086相比，80186新增加了哪些指令？它们的作用各是什么？

19. 80186和80188在哪些方面有区别？

第十五章　超级16位微处理器iAPX286/10

§15.1　概　　述

随着微型计算机技术和应用的飞速发展，人们日益增多地使用微处理器构成更大的系统，以便为操作系统提供更强有力的支持，大大降低软件开发的成本，Intel公司继8086,80186之

表15.1　执行时间

机　器	语　言	字	时间(毫秒)			
		size	search	sieve	puzzle	acker
VAX-11/780	C	32	1.4	250	9,400	4,600
	Pascal(UNIX)	32	1.6	220	11,900	7,800
	Pascal(VMX)	32	1.4	259	11,530	9,850
68000(8MHz)	C	32	4.7	740	37,100	7,800
	Pascal	16	5.3	810	32,470	11,480
	Pascal	32	5.8	960	33,520	12,320
68000(16MHz)	Pascal	16	1.3	196	9,180	2,750
	Pascal	32	1.5	246	9,200	3,080
8086(5MHz)	Pascal	16	7.2	764	44,000	11,100
80286(8MHz)	Pascal	16	1.4	168	9,138	2,218
80286(10MHz)	Pascal	16	1.1	135	7,311	1.774

表15.2　相对于VAX-11/780的性能

机　器	语　言	字	对VMX Pascal的比率(>1=>较快的)				
		size	search	sieve	puzzle	acker	avg+sd
VAX-11/780	C	32	1.0	1.0	1.2	2.1	1.3+ .4
	Pascal(UNIX)	32	.9	1.2	1.0	1.3	1.1+ .2
	Pascal(VMS)	32	1.0	1.0	1.0	1.0	1.0+ .0
68000(8MHz)	C	32	.3	.4	.3	1.3	.6+ .4
	Pascal	16	.27	.32	.36	.86	.5+ .2
	Pascal	32	.24	.27	.35	.80	.4+ .2
68000(16MHz)	Pascal	16	1.1	1.3	1.3	3.6	1.8+1.0
	Pascal	32	.95	1.0	1.3	3.2	1.6+ .9
8086(5MHz)	Pascal	16	.2	3	.3	.9	.4+ .3
80286(8MHz)	Pascal	16	1.0	1.5	1.3	4 4	2.1+1.4
80286(10MHz)	Pascal	16	1.3	1.9	1.6	5.6	2.6+1.7

表 15.3 在 8MHz 下的性能

等待状态	机器	语言	时间(毫秒)			
			search	sieve	puzzle$_{11}$	acker
4	68000	Pascal	5.3	810	32,470	11,480
0	8086	Pascal	4.6	448	27,500	6,938
	68000	Pascal	2.6	392	18,360	5,500
	80286	Pascal	1.4	168	9,138	2,218

后，在 80 年代初推出了一种以 80286 为 CPU 的具有存贮器管理和保护机构的新一代的 16 位微处理器——iAPX286。它的问世，标志着 16 位微处理器的发展，进入了又一个崭新的阶段，引起了各方面的极大关注。

比较往往是最能说明问题的。表 15.1——15.3 是 80286 和其他器件的性能比较，其中前两个表是相对于 VAX-11/780 所运行的 VMS PASCAL 的执行时间和性能的比较；表 15.3 则表示了在 8MHz 主频下的处理器性能比较。从中可以看到它的某些性能和指标已经接近或超过了某些典型的小型机。

为了把握住 16 位微型计算机的最新发展趋势，把具有代表性的 iAPX 286/10 作为典型，作一番分析研究是很有必要的。

§15.2 iAPX 286/10 概况

iAPX286，是指以超大规模集成电路芯片 80286，为 CPU 的微处理器系统，依据不同的应用，可以包含有不同的部件。iAPX286/10 结构是在 80286 CPU 的基础上，增加了时钟和接口电路。而 iAPX286/20 数值数据处理器(NDP)，则是在 iAPX 286/10 系统上增加一个 80287 NPX 构成的。

§15.2.1 iAPX 286/10 CPU

iAPX286/10，是一个以 80286 为 CPU 的先进的超级的微处理器，它具有对多用户和多任务系统的特别完善的处理能力。图 15.1 是 80286 CPU 的内部方框图。从中可以看到，80286 与 8086 相比增加了 IU，原来的 BIU 现在分成了 AU、IU 和 BU。图 15.2 表示了在 80286 内部并行操作的情况。

80286 具有集成在片内的存贮器管理机构，它能用四层特权支持操作系统和任务的分离，同样也能支持在任务中的程序和数据的保密。iAPX 286/10 提供的四层保护，可以用图 15.3 表示。10 MHz 的 iAPX 286/10，可以提供比 5MHz 的 iAPX86/10 大六倍的整体性能（见图 15.4）。

80286 具有很大的地址空间，并能以两种不同的方式运行。在实地址方式下，具有 1 兆字节的寻址能力；在虚地址保护方式下，能将每个任务的 2^{30} 字节虚地址映射到 2^{24} 字节的物理地址中去，也就是说，在这种方式下，具有 16 兆字节的寻址能力。图 15.5 表示了在两种不同方式下的地址映射关系。

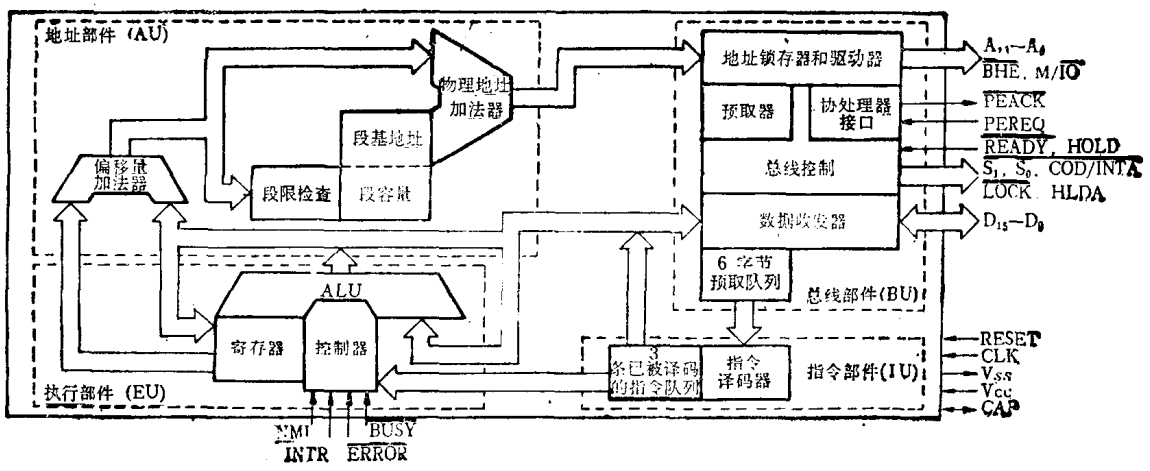
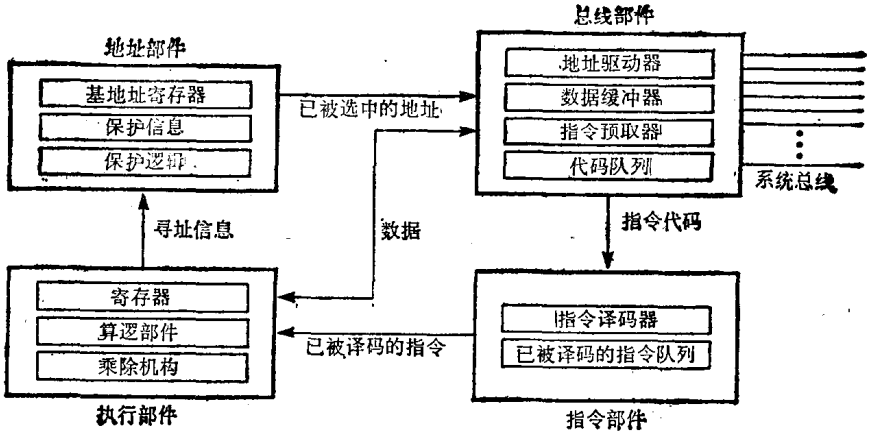
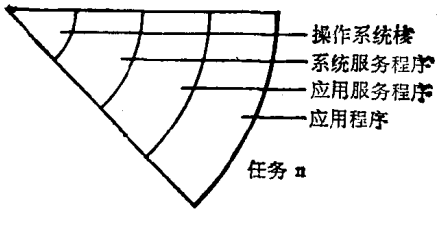


图 15.1 80286 内部方框图



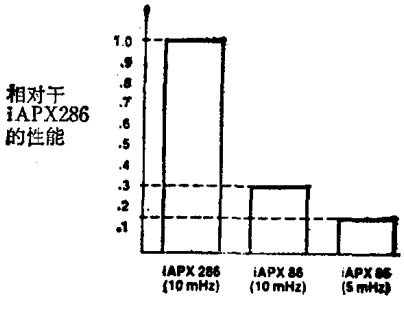
在 iAPX286 CPU 内部的四个处理部件并行地进行操作，提高了吞吐率

图 15.2 并行处理加快了速度



在 iAPX286 系统中的每一个任务的服务和应用程序，最多可以有四个分离的特权层

图 15.3 每个任务的四层特权



iAPX286 的性能可以是早期微处理器的六倍

图 15.4 iAPX286 性能比较

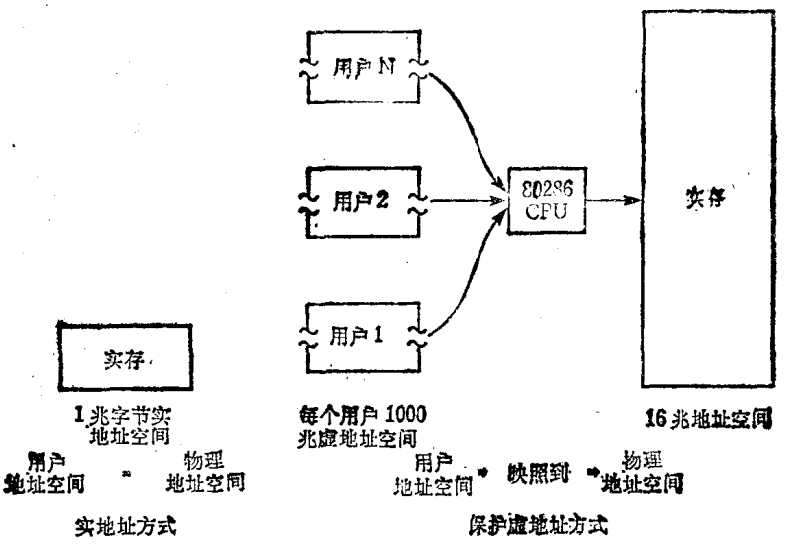


图 15.5　实地址方式和虚地址方式的地址空间

iAPX286 与 iAPX86 和 88，是软件向上兼容的。使用 iAPX 286 实地址方式的 80286 的目标代码、和已有的 iAPX86、88 的软件是兼容的；在保护虚地址方式下，80286 是和 iAPX 86、88 软件源代码相兼容的，这些源代码可以进一步升级，而使用由 80286 集成在片内的存贮器管理和保护机构支持的虚地址。无论是实地址方式还是虚地址方式，都能以 80286 的特性进行操作，并且执行一个 iAPX86 和 88 指令的高级集。

和 8086 一样，80286 可以通过附加任选的协处理器，来扩展系统的处理功能。例如，以 80286 和 80287 组合，来构成高性能的 80 位数值数据处理器 iAPX286/20。

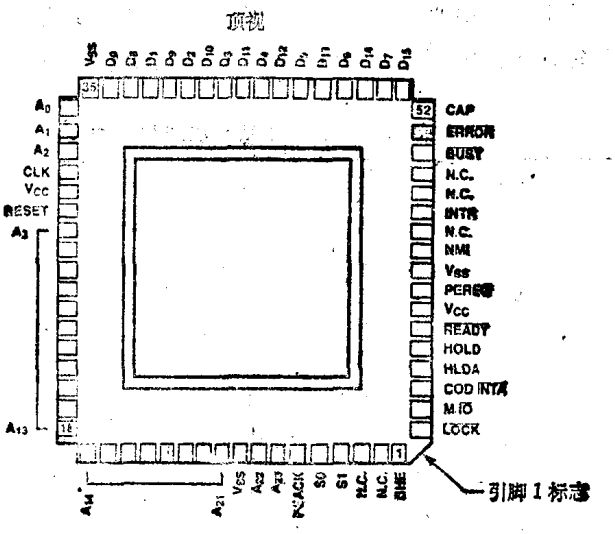
iAPX286/10 提供特殊的操作，来支持操作系统的有效实现和运行。例如，用一条指令就可以结束一个任务的执行，保护它的状态，转换到一个新的任务，装入新的状态，并且开始新任务的执行。iAPX 286/10 主要是通过提供段不存在异常和重新启动指令，来支持虚拟存贮系统的。

§15.2.2　80286 芯片引脚介绍

图 15.6 是 80286 芯片引脚图。从图中可见它已经突破了 8086 的 40 个引脚的格局，也不再采用地址数据总线和地址控制总线分时互锁的方式，而是具有独立的 24 根地址线和 16 根数据线。由于 80286 的性能，比之 8086 有了较大的提高，因而各引脚的功能与 8086 也不尽相同。现在把一些引脚的主要功能介绍如下；其中 I，表示该引脚上的信号是输入；O，表示该引脚上的信号是输出；I/O 表示该引脚既可用于输入又可用于输出。

(1) CLK I System Clock（系统时钟）第 31 脚

为 iAPX 286 系统提供基本定时。它是一个 16MHz 的信号，在 8086 内部被除以 2 而产生 8MHz 的处理器时钟。内部除以 2 电路，由在 RESET 输入引脚上的低电平到高电平的转变，来同步于一个外部时钟发生器。



注：N.C. 引脚不能被连接

图 15.6 80286 引脚

(2) D_{15}—D_0 I/O DataBus(数据总线)第 36—51 脚

在存贮器、I/O 和中断响应读周期中输入数据。在存贮器和 I/O 写周期中输出数据。数据总线是高电平有效，在总线保持响应期间，浮空到第 3 态 OFF。

(3) A_{23}—A_0 O Address Bus(地址总线)第 7,8,10—17,18—28,32—34 脚

输出物理存贮器和 I/O 接口地址。当数据要在引脚 D_7—D_0 上传送时，A_0 是低电平，在 I/O 传送时 A_{23}—A_{16} 是低电平。地址总线是高电平有效，在总线保持响应期间，浮空到第 3 态 OFF。

(4) $\overline{BHE}$ O Bus High Enable(总线高位允许)第 1 脚

该引脚的功能和 8086 的 $\overline{BHE}$ 引脚功能基本相同，它表示数据在数据总线的高字节 D_{15-8} 上进行传送。指定为数据总线高 8 位的设备，常常使用 $\overline{BHE}$ 作为片选信号。$\overline{BHE}$ 和 A_0 的编码如表 15.4 所示

表 15.4 $\overline{BHE}$ 和 A_0 的编码

$\overline{BHE}$	A_0	功能
0	0	字传送
0	1	在数据总线高 8 位进行字节传送 (D_{15-8})
1	0	在数据总线低 8 位进行字节传送 (D_{7-0})
1	1	保留

(5) $\overline{S_1}$,$\overline{S_0}$ O Bus Cycle Status(总线周期状态)第 4,5 脚

表示一个总线周期开始，并且和 M/$\overline{IO}$ 和 COD/$\overline{INTA}$ 信号一起定义总线周期的类型。一

且一个或两个信号都处于低电平，总线是处于 T_s 状态。$\overline{S_1}$、$\overline{S_0}$ 是低电平有效，在总线保持响应期间浮空到第 3 态 OFF。$\overline{S_1}$、$\overline{S_0}$ 和 COD/$\overline{INTA}$、M/$\overline{IO}$ 信号的编码如表 15.5 所示。

表 15.5 80286 总线周期状态定义

COC/$\overline{INTA}$	M/$\overline{IO}$	$\overline{S_1}$	$\overline{S_0}$	启 动 的 总 线 周 期
0	0	0	0	中断响应
0	0	0	1	保留
0	0	1	0	保留
0	0	1	1	不是一个状态周期
0	1	0	0	若 A1=1，则暂停，否则停机
0	1	0	1	读存贮器数据
0	1	1	0	写存贮器数据
0	1	1	1	不是一个状态周期
1	0	0	0	保留
1	0	0	1	读 I/O
1	0	1	0	写 I/O
1	0	1	1	不是一个状态周期
1	1	0	0	保留
1	1	0	1	读存贮器指令
1	1	1	0	保留
1	1	1	1	不是一个状态周期

(6) M/$\overline{IO}$ O Memory/IO Select（存贮器，IO 选择）第 67 脚

区分对存贮器和 I/O 的访问。在 T_s 期间若是高电平，则正处于一个存贮器周期或一个暂停/停机(halt/shutelown) 周期；若是低电平，则正处于一个 I/O 周期或中断响应周期。M/$\overline{IO}$ 在总线保持响应期间，浮空到第 3 态 OFF。

(7) COD/$\overline{INTA}$ O Code/Interrupt Acknowledge（代码/中断响应）第 66 脚

区分取指令周期和读数据周期，也区分中断响应周期和 I/O 周期，在总线保持响应期间，浮空到第 3 态 OFF。

(8) $\overline{LOCK}$ O Bus LOCK（总线封锁）第 68 脚

表示在当前的总线周期后，其它的系统总线主设备不能取得系统总线的控制权。$\overline{LOCK}$ 信号可以直接由 "LOCK" 指令前缀激活，或由 80286 的硬件在存贮器 XCHG 指令、中断响应或描述子表访问期间自动产生。$\overline{LOCK}$ 是低电平有效，并且在总线保持响应期间，浮空到第 3 态 OFF。

(9) $\overline{READY}$ I Bus Ready（总线准备就绪）第 63 脚

结束一个总线周期。一个总线周期在被 $\overline{READY}$ 低电平终止以前，将无限地扩展。$\overline{READY}$ 是一个低电平有效的同步输入，它请求建立和保持与系统时钟有关的时间，以满足正确操作的需要。在总线保持响应期间，$\overline{READY}$ 被忽略。

(10) HOLD I Bus Hold Request and Hold Acknowledge HLDA O
(总线保持请求和保持响应)第 64,65 脚

控制 80286 局部总线所有权。HOLD 输入允许另外一个局部总线主设备请求对局部总线的控制。当转让控制时，80286 将把它的总线驱动器浮空到第 3 态 OFF，并激活 HLDA。这样，就进入了总线保持响应状态。该局部总线将保持转让给提出请求的主设备，直到 HOLD 变成无效。HOLD 的变化，使 80286 撤消 HLDA，并重新取得局部总线的控制权，这样，就终止了总线保持响应状态。HOLD 可以异步于系统时钟。这些信号高电平有效。

(11) INTR I Interrupt Request(中断请求)第 57 脚

请求 80286 挂起当前的程序执行，并且为一个外部的急迫的请求服务。一旦在标志字中的允许中断位被清除，中断请求就被屏蔽。当 80286 响应一个中断请求时，它执行两个中断响应总线周期，以读得一个识别中断源的 8 位中断向量。为了确保程序中断，INTR 必须保持有效，直到第一个中断响应周期完成。INTR 被在每个处理器周期的开头取样，并且在当前指令结束之前，至少保持 2 个高电平有效的处理器周期，以便能在下一条指令之前实现中断。INTR 是电平感应的，高电平有效，并且可以异步于系统时钟。

(12) NMI I Non-maskable Interrupt Request(不可屏蔽中断请求)第 59 脚

用一个内部提供的向量值 2 中断 80286，不执行中断响应周期。在 80286 标志字中的中断允许标志位，不影响这种中断。NMI 输入是高电平有效，可以异步于系统时钟，并且是在内部同步后由边缘触发的。为了正确地识别，该输入必须至少前 4 个系统时钟周期是低电平，并且保持至少 4 个系统时钟周期的高电平。

(13) PEREQ I Processor Extension Operand and Acknowledge $\overline{\text{PEACK}}$ O
(协处理器操作数请求和响应)第 61,6 脚

把 80286 的存贮器管理和保护能力，扩展到协处理器。PEREQ 输入请求 80286 为一个协处理器执行一个数据操作数传送。PEACK 在被请求的操作数正被传送时通知协处理器。PEREQ 高电平有效，并且可以异步于系统时钟。$\overline{\text{PEACK}}$ 是低电平有效。

(14) $\overline{\text{BUSY}}$ I Processor Extension Busy and Error $\overline{\text{ERROR}}$ I
(协处理器忙和出错)第 54,53 脚

为 80286 指示协处理器的操作情况。有效的 $\overline{\text{BUSY}}$ 输入将停止 80286 程序在 WAIT 和一些 ESC 指令上的执行，直到 BUSY 变成无效(高电平)。在等待 BUSY 变成无效期间，80286 可以被中断。有效的 ERROR 输入将使 80286 在执行 WAIT、或一些 ESC 指令时，实现协处理器的中断。这些输入是低电平有效，并且能异步于系统时钟。

(15) RESET I System Reset(系统复位)
第 29 脚

清 80286 的内部逻辑，并且是高电平有效。80286 在任何时候，都可以用在 RESET 脚上

从低电平到高电平的转换来重新初始化。这个高电平应该保持有效多于 16 个系统时钟周期。在 RESET 有效期间，80286 的输出引脚进入如表 15.6 所示的状态。

表 15.6 RESET 有效期间输出引脚的状态

引脚状态	引脚名
1（高电平）	$\overline{S_0}$, $\overline{S_1}$, PEACK, A_{23}—A_0, $\overline{BHE}$, $\overline{LOCK}$
0（低电平）	M/$\overline{IO}$, COD/$\overline{INTA}$, HLDA
3 态 OFF	D_{15}—D_0

80286 的操作是在RESET 脚上由高电平到低电平的转换之后开始。RESET 的高电平到低电平的转换，必须同步于系统时钟。在执行第一个从接通电源后的运行地址中取代码的总线周期之前，80286 为了内部初始化，大约需要 50 个时钟周期。

RESET 同步于系统时钟的从高电平到低电平的转换，将在系统时钟的下一个高电平到低电平的转换时，开始一个新的处理器周期。RESET 从低电平到高电平的转换，也可以异步于系统时钟，然而，在这种情况下，它不能预先确定在下一个系统时钟期间，处理器时钟将会出现那一个节拍。RESET 的同步的低电平到高电平的转换，只为那些处理器时钟必须节拍同步于另一个时钟的系统所需要。

(16) V_{SS} I System Ground(系统地)

第 30 脚

0 伏

(17) V_{CC} I System Power(系统电源)

第 60 脚

+5 伏电源

(18) CAP I Substrate Filer Capacitor(衬底滤波电容器)第 52 脚

在该引脚和地之间，必须接一个 0.047μf±20% 的电容器。该电容器用来滤内部衬底偏压发生器的输出，该电容器允许有 1μA 的漏电流。

为了使 80286 能正确地操作，衬底偏压发生器必须把这个电容充电到它的工作电压。在 V_{CC} 和 CLK 到达它们额定的 AC 和 DC 参数以后，这个电容的充电时间为 5ms（最大），在这期间，RESET 可以用来防止 CPU 产生假动作。在这之后，通过产生同步于系统时钟的 RESET LOW 脉冲，80286 处理器时钟可以节拍同步于另一个时钟。

§15.3 iAPX 286/10 基本结构

从前面的介绍中，已经可以粗略地看到 80286 具有比 8086 强得多的功能。为了获得较强的功能，它必须要有较强的硬件支持能力。为了能与以前的产品相兼容，它的寄存器结构和指令集，必须是原来产品的母集，80286 作为 iAPX 286 的 CPU 正是这样的片子。iAPX286 的实地址方式操作，基本上兼容了 8086 的所有操作；iAPX286 的虚地址保护方式的操作，则提

供了许多新的扩充的功能。下面我们先介绍对两种方式是共同的80286的基本结构，然后介绍iAPX286的实地址方式、最后介绍虚地址保护方式。在介绍中，特别是在实地址方式中，与8086相同的就尽量从略(读者可以参阅本书上册中的有关章节)。

§15.3.1 寄存器集

80286的基本结构，具有15个寄存器，如图15.7所示。它们可以分成4组，其中通用寄存器、段寄存器、基址和变址寄存器与8086完全一样，状态和控制寄存器则有些变化，主要表现在标志位增加了3位，并且新增了一个MSW寄存器。现分述如下：

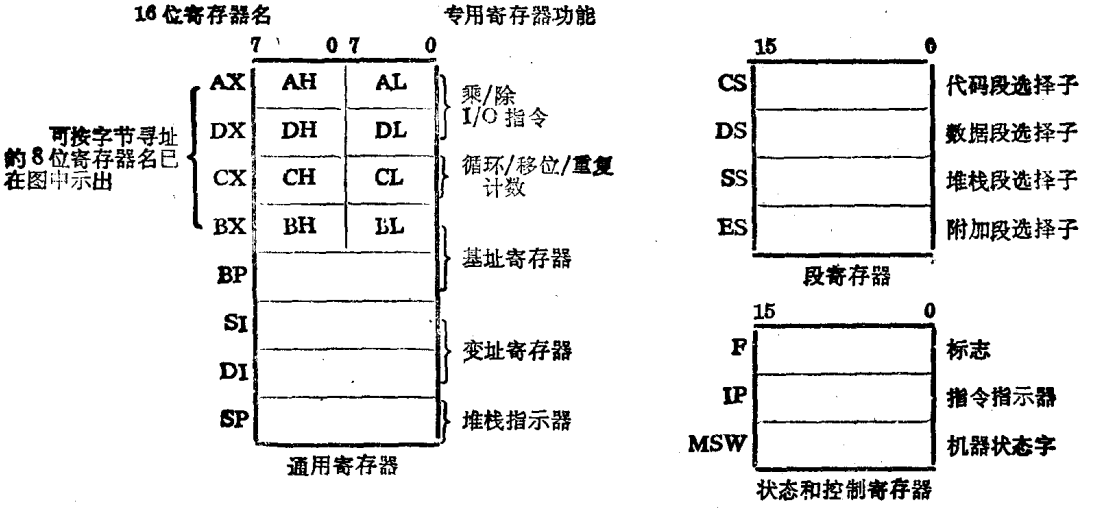


图15.7 基本寄存器集

1. 通用寄存器

8个16位通用寄存器，用于存放算术、逻辑操作数。其中4个寄存器AX、DX、CX、BX既可作为16位的字寄存器，也可作为两个分离的8位字节寄存器对。

2. 段寄存器

4个16位专用寄存器，可在任何给定的时刻，选择立即寻址，分别对应于代码、堆栈和数据的存贮器段(详见存贮器组成部分)。

3. 基址和变址寄存器

通用寄存器中的4个寄存器，也可以用来确定操作数在存贮器中的偏移地址。这些寄存器含有在一个段中特定单元的基地址、或变址值。哪一个寄存器参加确定操作数地址计算，则由寻址方式确定。

4. 状态和控制寄存器

3个16位专用寄存器，记录或控制80286处理器的某些方面的状态，其中包括指令指示

器(IP),它用于指出要执行的下一顺序指令的偏移地址。

§15.3.2 iAPX286/10 标志字和机器状态字

iAPX286/10 的标志字和机器状态字的位功能,由图 15.8 来表示。其中标志字反映了算术和逻辑指令的结果的性质,80286 的操作方式,以及在进行 I/O 时该 I/O 操作所在的特权层。机器状态字,则记录了当前处理器所处的状态。标志字的功能由表 15.7 给出;机器状态字的功能由表 15.8 给出。

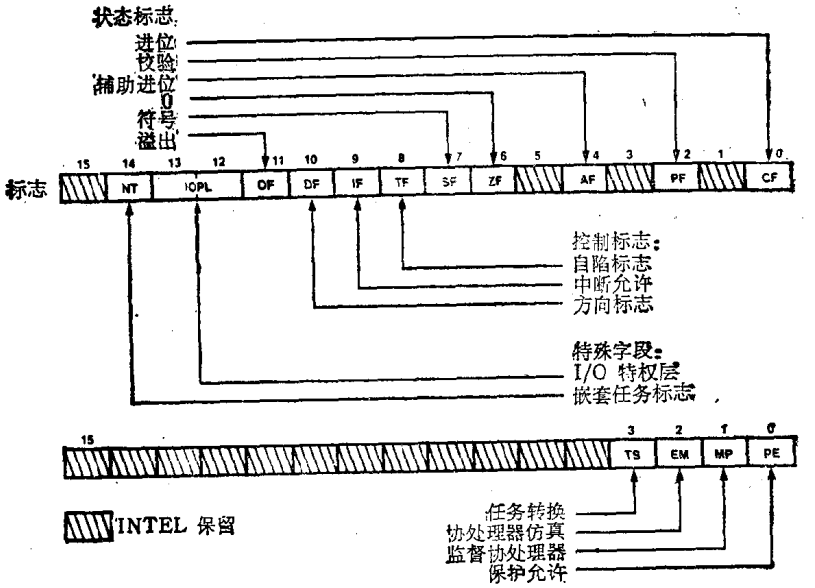


图 15.8 标志字和机器状态字的位功能

表 15.7 标志字功能

位	名 称	功 能
0	CF	进位标志:当最高位出现进位或借位时,CF=1,否则 CF=0
2	PF	奇偶校验标志:当低 8 位结果含有偶数个 1 时 PF=1,否则 PF=0
4	AF	辅助进位标志:当 AL 的低 4 位出现向高位的进位或借位时 AF=1,否则 AF=0
6	ZF	零标志:当结果为 0 时,ZF=1,否则 ZF=0
7	SF	符号标志:SF=结果的最高位(0 表示正,1 表示负)

位	名　称	功　能
11	OF	溢出标志：当结果对于目操作数是一太大的正数或太小的负数（包括符号位）时，OF=1，否则 OF=0
8	TF	单步标志：一旦 TF=1，则在下一条指令执行完之后，产生单步中断，TF 标志将由单步中断处理程序清 0。
9	IF	中断允许标志：当 IF=1 时，处理器可以被 INTR 引脚上出现的信号所中断，并把控制转移到中断向量所指定的单元。
10	DF	方向标志：当 DF=1 时，串指令自动增量指定的变址寄存器，当 DF=0 时，串指令自动减量指定的变址寄存器。
12—13	IOPL	I/O 特权标志：指定 I/O 操作处于 0—3 特权层中的那一层。
14	NT	嵌套任务标志：当 NT=1 时表示当前执行的任务嵌套于另一任务中，执行完该任务后，要返回到原来的任务中去，否则 NT=0

表 15.8　机器状态字位功能

位	名　称	功　能
0	PE	保护方式允许：把 80286 放入保护方式，并且除 RESET 外不能被清除。
1	MP	监督协处理器：允许 WAIT 指令引起一个协处理器不存在异常(7 号)。
2	EM	仿真协处理器：当 ESC 指令允许仿真一个协处理器时，将引起一个协处理器不存在异常(7 号)。
3	TS	任务转换：表示下一条要使用一个协处理器的指令将会引起异常 7，允许用软件测试当前协处理器是否上下文属于当前的任务。

机器状态字(MSW)，是一个 16 位寄存器，目前使用的是它的低 4 位。其中一位用来使 CPU 进入保护方式，其他三位则起控制协处理器接口的作用。在 RESET 之后，该寄存器包含 FFF0H，它把 80286 放入 iAPX286 的实地址方式。LMSW 和 SMSW 指令可以在实地址方式下装入和存放 MSW。关于 TS、EM 和 MP 的编码如表 15.9 所示。

表 15.9　MSW 中控制协处理器的位的编码

TS	MP	EM	意　义	造成异常的指令
0	0	0	仅是 iAPX 286 的实地址方式。在 RESET 之后的最初编码。iAPX 286 的操作与 iAPX86,88 相同。	无
0	0	1	没有协处理器可供使用，软件将仿真它的功能	ESC
1	0	1	没有协处理器可供使用，软件将仿真它的功能。当前协处理器的上下文可以属于另一个任务。	ESC
0	1	0	协处理器存在。	无
1	1	0	协处理器存在。当前协处理器的上下文可以属于另一个任务。关于 WAIT 的异常允许软件检测来自前一个协处理器操作期间的错误。	ESC 或 WAIT

§15.3.3 iAPX 286/10 指令集

iAPX286/10 的指令集，可以分为七类，即数据传送；算术运算；移位/环移/逻辑；串操作；控制转移；高级指令以及处理器控制指令。

80286 的指令能引用 0 个、1 个或 2 个操作数。这些操作数可以在寄存器中，也可以在存贮器中、或在指令中。0 操作数指令（如 NOP 和 HLT）通常是单字节长。单操作数指令（如 INC 和 DEC）通常是 2 字节长，但其中一些仅在 1 字节中编码。单操作数指令可以引用一个寄存器、或存贮器单元。双操作数指令可以对操作数进行以下 6 种处理：

(1) 寄存器到寄存器；
(2) 存贮器到寄存器；
(3) 立即数到寄存器；
(4) 存贮器到存贮器；
(5) 寄存器到存贮器；
(6) 立即数到存贮器。

双操作数指令（如 MOV 和 ADD），通常是 3～6 字节长。存贮器到存贮器，由需要 1—3 个字节的专门类型的串指令提供，指令的详细格式和编码，可参见本章末尾的 80286 指令总结。

图 15.9 表示了各种类型的指令。

通用指令	
MOV	传送字节或字
PUSH	把字推入到堆栈
POP	把字从堆栈中弹出
PUSHA	把所有的寄存器推入堆栈
POPA	把所有的寄存器值从堆栈中弹出
XCHG	交换字节或字
XLAT	翻译字节
输入/输出	
IN	输入字节或字
OUT	输出字节或字
地址目标	
LEA	装入有效地址
LDS	使用 DS 装入指示器
LES	使用 ES 装入指示器
标志传送	
LAHF	从标志字中装入 AH 寄存器
SAHF	把 AH 寄存器内容存放入标志
PUSHF	把标志推入堆栈
POPF	把标志从堆栈中弹出

图 15.9(a) 数据传送指令

MOVS	传送字节或字串
INS	输入字节或字串
OUTS	输出字节或字串
CMPS	比较字节或字串
SCAS	扫描字节或字串
LODS	装入字节或字串
STOS	存放字节或字串
REP	重复
REPE/REPZ	在相等/为0时重复
REPNE/REPNZ	在不相等/不为0时重复

图 15.9(b) 串指令

加　　法

ADD	字节或字相加
ADC	字节或字带进位相加
INC	把字节或字加1
AAA	加法ASCII调整
DAA	加法10进制调整

减　　法

SUB	字节或字相减
SBB	字节或字带借位相减
DEC	字节或字减1
NEG	把字节或字变成相反数
AAS	减法ASCII调整
DAS	减法10进制调整

乘　　法

MUL	无符号字或字节相乘
IMUL	字节或字整数乘
AAM	乘法ASCII调整

除　　法

DIV	无符号字节或字相除
IDIV	字节或字整数除
AAD	除法ASCII调整
CBW	字节到字转换
CWD	字到双字转换

图 15.9(c) 算术指令

逻　　辑	
NOT	“非”字节或字
AND	“与”字节或字
OR	“或”字节或字
XOR	“异或”字节或字
TEST	“测试”字节或字
移　　位	
SHL/SAL	把字节或字逻辑/算术左移
SHR	把字节或字逻辑右移
SAR	把字节或字算术右移
环　　移	
RCL	把字节或字左环移
ROR	把字节或字右环移
RCL	把字节或字通过进位位左环移
RCR	把字节或字通过进位位右环移

图 15.9(d)　移位/环移/逻辑指令

条　件　转　移	
JA/JNBE	若高于/不低于等于，则跳转
JAE/JNB	若高于等于/不低于，则跳转
JB/JNAE	若低于/不高于等于，则跳转
JBE/JNA	若低于等于/不高于，则跳转
JC	若有进位，则跳转
JE/JZ	若相等/为 0，则跳转
JG/JNLE	若大于/不小于等于，则跳转
JGE/JNL	若大于等于/不小于，则跳转
JL/JNGE	若小于/不大于等于，则跳转
JLE/JNG	若小于等于/不大于，则跳转
JNC	若无进位，则跳转
JNE/JNZ	若不相等/不为 0，则跳转
JNO	若不溢出，则跳转
JNP/JPO	若校验为非偶/校验为奇，则跳转
JNS	若无符号，则跳转
JO	若溢出，则跳转
JP/JPE	若校验为非奇/校验为偶，则跳转
JS	若无符号，则跳转
无　条　件　转　移	
CALL	调用过程
RET	从过程中返回
JMP	跳转
重　复　控　制	
LOOP	循环
LOOPE/LOOPZ	若相等/为零，则循环
LOOPNE/LOOPNZ	若不相等/不为零
JCXZ	若寄存器 CX=0，则跳转

中　　断	
INT	中断
INTO	若溢出，就中断
IRET	中断返回

图 15.9(e)　程序转移指令

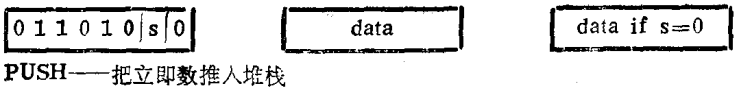
标　志　操　作	
STC	设置进位标志
CLC	清除进位标志
CMC	把进位标志取补
STD	设置方向标志
CLD	清除方向标志
STI	设置中断允许标志
CLI	清除中断允许标志
外　部　同　步	
HLT	暂停，直到中断或复位
WAIT	等待 TEST 引脚有效
ESC	交权给协处理器
LOCK	在下一条指令期间封锁总线
空　　操　　作	
NOP	无操作
执　行　环　境　控　制	
LMSW	装入机器状态字
SMSW	存放机器状态字

图 15.9(f)　处理器控制指令

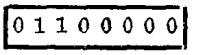
ENTER	为过程入口格式化堆栈
LEAVE	为过程终止恢复堆栈
BOUND	检察预定范围之外的值

图 15.9(g)　高级指令

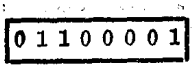
图 15.9 列出了实地址方式，和保护虚地址方式公用的指令，从中可以看到绝大部指令是和 8086 指令相同的，但在数据传送类中增加了 PUSHA 和 POPA 指令，有的指令则增加了操作方式；在串处理类中，增加了 INS 和 OUTS 指令，以实现成批数据的输入/输出和移动；在控制转移类中，增加了 ENTER、LEAVE 和 BOUND 等 3 条有关过程处理的高级指令。80286 还有在图 15.9 中没有列出的保护控制类 15 条新指令，这些指令完全是对 8086 指令的扩充。为了使读者对 80286 指令集新增加的功能和指令，有一个比较全面的了解，特把这些有关指令总结在图 15.10 中。



PUSH——把立即数推入堆栈



PUSHA——把所有的寄存器推入堆栈

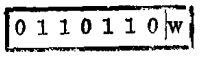


POPA——从堆栈中弹出所有寄存器

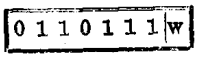
IMUL——带符号整数乘立即数

Shift/Rotate——带记数的移位指令

TTT	指令
000	ROL
001	ROR
010	RCL
011	RCR
100	SHL/SAL
TTT	指令
101	SHR
111	SAR



INS——从DX指定的转接口中输入字节/字串

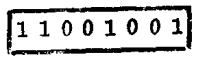


OUTS——在DX指定的转接口中输出字节/字串

INS——输入字节或字串

OUTS——输出字节或字串

ENTER——进入过程

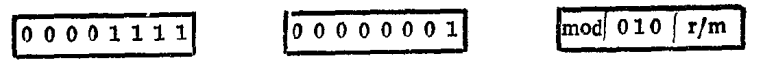


LEAVE——离开过程

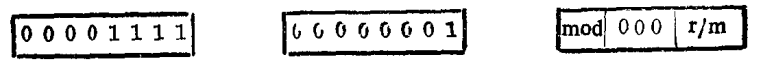
BOUND——检测超出范围的值

CTS——清任务转换标志

保护控制指令



LGDT——装入全局描述子表寄存器



SGDT——存放全局描述子表寄存器

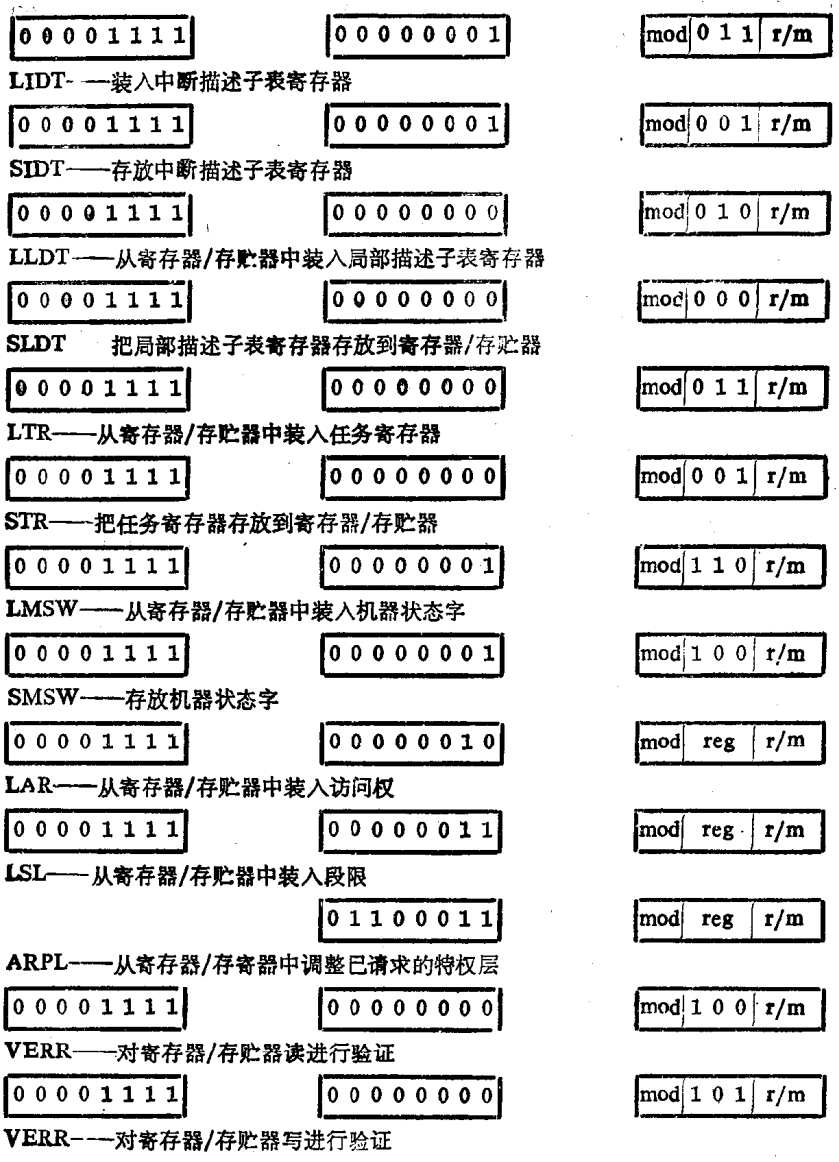


图 15.10 80286 新增加的指令和指令功能

§15 3.4 存贮器组成

iAPX286 的存贮器被组成可变长度的段，每一个段是多至 64K 的线性相邻字节序列. 存贮器使用具有两个分量的地址指示器来寻址，这两个分量分别为一个 16 位的段选择子，和一个 16 位的偏移量. 其中段选择子，用来选择在存贮器中的所需要的段，偏移量则表示在该段内所需要的字节地址. 图 15.11 表示了指示器两个分量的含义.

所有在存贮器中寻址操作数的指令，必须指明该操作数在存贮器的哪一段，和在该段中的偏移量. 为了使指令执行起来更快和使指令编码更加紧凑，段选择子通常存放在一个高速寄存器中，这样，为了寻址一个存贮器操作数，一条指令只要指明所要求的段寄存器和在该段中的偏移量就可以了.

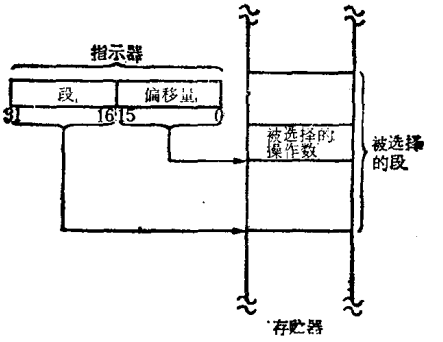


图 15.11 构成存贮器地址的两个指示器分量

大多数指令和 8086 的指令一样，不需要直接指明使用的是哪一个段寄存器。隐含的段寄存器使用规则如表 15.10 所示。这些规则适用于这样一种写程序的方法，即把程序写成为独立的模块（见图 15.12），各模块要求不同的代码、数据、堆栈以及对外部数据进行访问的区域。

表 15.10 段寄存器选择规则

需要的存贮器引用	使用的段寄存器	隐含的段选择规则
指令	Code(CS)	自动地用于指令的预取
堆栈	Stack(SS)	所有的堆栈的推入和弹出操作，任何把 BP 作为基地址寄存器使用的存贮器引用
局部数据	Data(DS)	除了关于堆栈或串目之外的所有数据引用
外部(全局)数据	Extra(ES)	替换数据段和串操作目

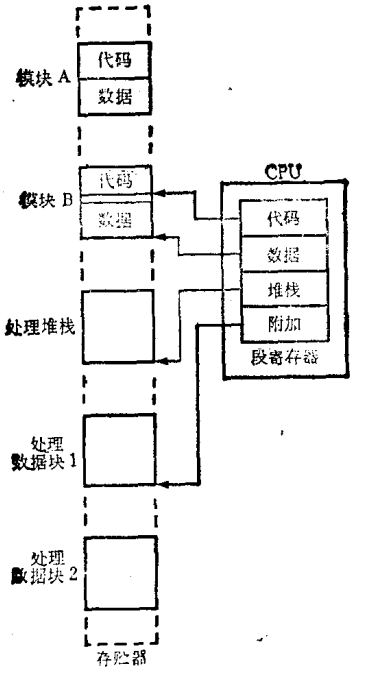


图 15.12 分段的存贮器有利于结构化的软件设计

专用的段超越指令前缀，使处理器在某些特殊情况下超越某些段寄存器选择规则。对于一个单个的程序、堆栈、数据和附加段可以是重叠的。要访问一个不驻留在4个当前可用段中的操作数时，则要使用一个32位的指示器，或装入一个新的选择子。

§15.3.5 寻址方式

80286提供了总共8种指令操作数寻址方式。其中两种方式是用于处理寄存器或立即操作数指令的，它们是：

寄存器操作数方式：操作数包含在某个8位或16位通用寄存器中；

立即操作数方式：操作数包含在指令本身中。

其他六种方式，用来指明操作数在一个存贮器段中的地址(即偏移量)。前面已经介绍过，一个存贮器操作数的地址，是由两个16位分量组成，即段选择子和偏移量。段选择子是由段寄存器提供的，该段寄存器既可由寻址方式隐含地选择，也可用段超越前缀直接选择。同8086一样，80286的地址偏移量也是由3个地址分量的任意组合之和形成的，这3个分量如下：

(1) 位移量(包含在指令中的8位或16位的立即数)；

(2) 基址值(BX或BP基址寄存器的内容)；

(3) 变址值(SI或DI变址寄存器的内容)；

在地址计算中，由16位加法所产生的进位均被忽略。8位的位移量，在运算中要被符号扩展成16位。

上述的3个地址分量的组合，定义了6种存贮器寻址方式，即：

(1) 直接寻址方式：操作数的偏移量直接包含在指令中，作为8位或16位的位移量元素；

(2) 寄存器间接方式：操作数的偏移量在寄存器SI、DI、BX或BP之一中；

(3) 基址寻址方式：操作数的偏移量，是一个8位、或16位的位移量与基址寄存器(BX或BP)的内容之和；

(4) 变址寻址方式，操作数的偏移量，是一个8位、或16位的位移量与变址寄存器(SI或DI)的内容之和；

(5) 基址变址方式：操作数的偏移量，是一个基址寄存器和一个变址寄存器的内容之和；

(6) 具有位移量的基址变址方式：操作数的偏移量，是一个基址寄存器的内容，一个变址寄存器的内容和一个8位、或16位的位移量之和。

80286各种寻址方式的实现，可参阅本书上册对8086寻址方式的论述。

§15.3.6 数据类型

80286直接支持下列数据类型：

整型：包含在8位字节、或16位字中的带符号2进制数。所有的操作规定用2的补码表示，其中带符号的32位和64位整数，是由iAPX286/20数值数据处理器支持的。

序数：包含在一个8位字节、或16位字中的无符号的2进制数值。

指示器：一个 32 位的数量，由各为 16 位的一个段选择子分量、和一个偏移量分量组成。

串：一个相邻接的字节、或字的序列。一个串含有 1 至 64K 字节。

ASCII：字母数字的一字节表示和用字符的 ASCII 标准表示的控制字符。

BCD：10 进制数字 0—9 的 1 字节表示(非压缩)。

压缩 BCD：2 个 10 进制 0—9 的数字的 1 字节表示，其中每个数字各占 4 位。

浮点数：带符号的 32、64 或 80 位实数。浮点操作数由 iAPX286/20 数值数据处理器支持。

图 15.13 表示了 iAPX286 支持的数据类型

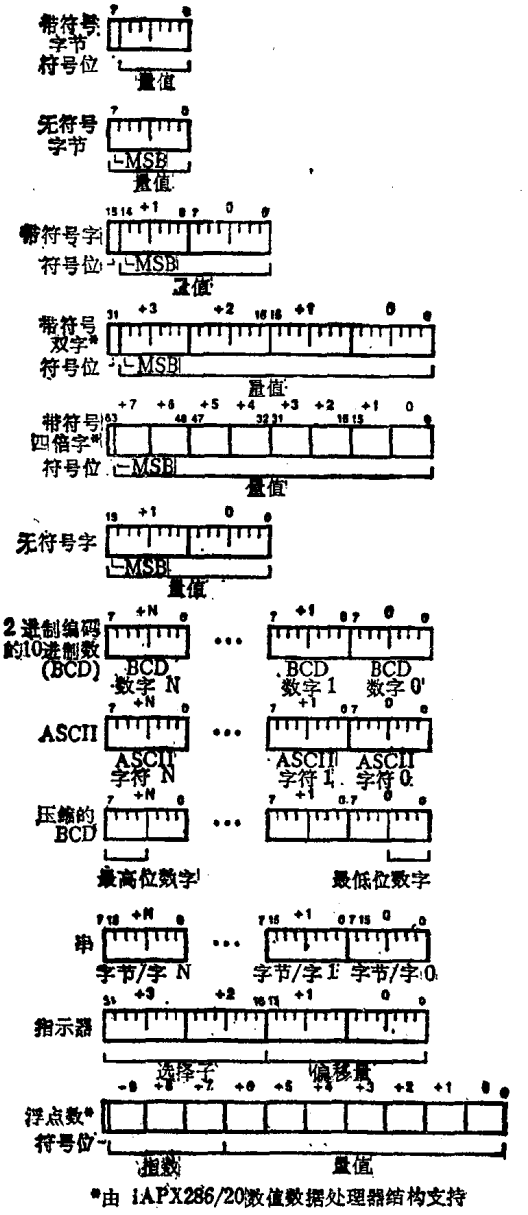


图 15.13 iAPX286 支持的数据类型

§15.3.7 I/O 空　间

I/O空间，是由64K 8位转接口、或32K 16位转接口所组成。I/O指令可以用一个在指令中指明的8位转接口地址来寻址I/O空间，也可以用放在DX寄存器中的16位转接口地址来寻址。8位转接口地址必须0扩展为16位，在这种情况下，$A_{15\sim8}$为低电平。I/O转接口地址00F8H到00FFH是保留的。

§15.3.8 中　断

iAPX286的中断可以分成3种类型：

(1) 硬件引起的；

(2) INT指令引起的；

(3) 指令异常引起的。

其中硬件引起的中断，产生于外部的中断请求，这种中断请求又被分成不可屏蔽中断和可屏蔽中断两类。程序可以用INT指令产生中断。指令异常在一种异常情况下发生，它能防止进一步执行后续的指令。异常情况是在试图执行一条指令时被检测到的。异常中断的作用，简单地说，就是在执行一条指令的条件还不具备时，通过异常中断处理，使这种条件具备，从而使指令能正常地执行。例如，指令要引用一个不在内存中的操作数，这时就会产生一个异常中断通过异常中断处理，把该操作数所在的段，从外存中调入内存，然后再启动该指令执行。所以一个异常中断返回的地址，总是指着引起该异常的指令，并且包括可能在其前面的任何指令前缀。

iAPX286系统含有一个包含多至256个指示器的表，为每一个中断定义专用的中断服务程序。中断类型0—31是为系统本身保留的，其中一些用于指令异常。对于每个中断，必须把一个8位的向量提供给80286，80286用该向量在表中寻找正确的表项。异常的中断向量，是在内部提供的。INT指令隐含中断向量3，或含有中断向量，并且能访问所有的256种中断。可屏蔽的硬件引起的中断，在一个中断响应总线序列中，把8位中断向量提供给CPU。不可屏蔽的硬件中断，使用事先定义的内部提供的向量。表15.11列出了中断向量分配的情况。

1．可屏蔽中断(INTR)

80286提供了一个可屏蔽硬件中断请求引脚INTR。当软件设置标志字中的中断标志位IF时，该引脚上的输入有效。所有224个用户定义的中断源，能共用这一输入，但它们各自的中断处理，却是互不相同的。当正在为一个中断服务时，可以把复位IF作为响应中断、或异常的一部分操作，而使更多的可屏蔽中断无效。被保护的标志字，将反映中断前处理器的中断允许状态。除非在中断处理程序中，有专门的指令重新设置IF标志，否则该中断标志将一直是0，直到被保护的标志字重新存放到标志寄存器为止。中断返回指令包括重新存入标志字操作，所以通过执行中断返回指令，就恢复了IF的原始状态。

2．不可屏蔽中断(NMI)

80286还提供了一个不可屏蔽中断输入引脚NMI，其优先级高于INTR。NMI具有代表性的用途，是启动电源掉电程序。这个输入将引起一个具有内部提供的向量值为2的中断，这种中断将不执行任何外部中断响应序列。

表 15.11 iAPX286 中断向量分配

功　　能	中断号	相 关 的 指 令	返回引起异常指令前的地址吗?
除法出错异常	0	DIV IDIV	返　回
单步中断	1	所有指令	
NMI 中断	2	所有指令	
断点中断	3	INT	
INTO 检测到溢出异常	4	INTO	不返回
BOUND 范围越出异常	5	BOUND	返　回
无效操作码异常	6	任何没有定义的操作码	返　回
协处理器不可用异常	7	ESC 或 WAIT	返　回
保留	8—15		
协处理器出错中断	16	ESC 或 WAIT	
保留	17—31		
用户定义	32—255		

当 80286 正在执行一个 NMI 服务程序时，它将不再为其他的 NMI 请求和 INTR 请求服务，也不为协处理器的段超越中断服务，这种状态要一直延续到执行中断返回(IRET)指令或 CPU 被复位(RESET)为止。若正在为某一 NMI 服务时，发生了另一 NMI 请求，则这种请求将被保留，以便在第一条 IRET 指令执行之后再进行服务。在 NMI 中断开始时，IF 位就被清除，以禁止 INTR 中断。

3. 单步中断

80286 具有一种内部中断，以使程序每次只执行一条指令，因此被称为单步中断。这种中断是由标志字中的单步中断标志(TF)位来控制的。一旦这个标志被置位，则在下一条指令执行完毕后，产生一个内部的单步中断。这种中断将清除 TF 标志位，并使用一个由内部提供的值为 1 的中断向量。在中断处理完之后，执行 IRET 指令，就恢复了被保护的 TF 标志，从而把控制转移到了下一条要单步执行的指令上。

§15.3.9 中断优先级

在同时发生若干种中断请求时，80286 将按表 15.12 中所给出的固定顺序对它们进行处理。中断处理包括保护标志字、返回地址，并把指向中断处理程序的第一条指令的指示器装入

表 15.12 中断处理顺序

顺　　序	中断
1	INT 指令或异常
2	单步
3	NMI
4	协处理器段超越
5	INTR

CS:IP 寄存器。若允许发生其他的中断，则它们将在当前中断处理程序的第一条指令被执行之前，进行处理。因此，最后处理的中断，最先得到服务。

§15.3.10 初始化和处理器复位

处理器的初始化或启动，是通过把 RESET 输入引脚驱动为高电平来实现的。RESET 将强迫 80286 结束所有的操作和局部总线的活动。只要 RESET 有效，就不会执行指令和出现总线活动。在 RESET 失效后，经过一段时间的内部处理以后，80286 开始用在物理位置 0FFFFF0H 的指令以实地址方式执行。表 15.13 列出了 RESET 为一些寄存器设置的预先定义的值。

表 15.13 在 RESET 之后 80286 寄存器的初始态状

标志字	0002H
机器状态字	FFF0H
指令指示器	FFF0H
代码段寄存器	F000H
数据段寄存器	0000H
附加段寄存器	0000H
堆栈段寄存器	0000H

§15.3.11 暂 停

HLT 指令暂停程序的执行，并且能在暂停期间防止 CPU 使用局部总线直至重新启动。NMI 引脚上的信号、IF = 1 时 INTR 引脚上的信号以及 RESET 将迫使 80286 退出暂停状态。若发生中断，则被保护的 CS:IP 的内容是指向 HLT 指令以后的一条指令的，中断结束后恢复被保护的 CS:IP，处理器就退出暂停状态而恢复执行。

§15.3.12 扩 展 能 力

前面说明了 80286 的基本结构所具有的寄存器、数据类型、寻址方式和指令。为了扩展这一基本结构，Intel 公司推出了 80287 数值协处理器(NPX)，它提供了浮点寄存器、多精度数值数据类型和一些新的算术指令。下面对它们作一些简单的介绍，以使读者对 iAPX286/10 加上 80287 构成的 iAPX286/20 数值数据处理器的性能有一个大概的了解。

1．扩展了寄存器集

扩展的寄存器集，包含 8 个 80 位的浮点寄存器，它能提供相当于 40 个 16 位寄存器的容量。图 15.14 表示了扩展的寄存器。这 8 个浮点寄存器是面向堆栈的，以简化算术程序设计。两个 16 位寄存器控制和报告数值指令的结果。控制字寄存器，定义了 IEEE 标准所要求的舍入、无穷大精度以及错误屏蔽控制。状态字寄存器，报告在数值操作中，所检测到的任何错误，它还包含有一个条件代码以控制条件分枝。

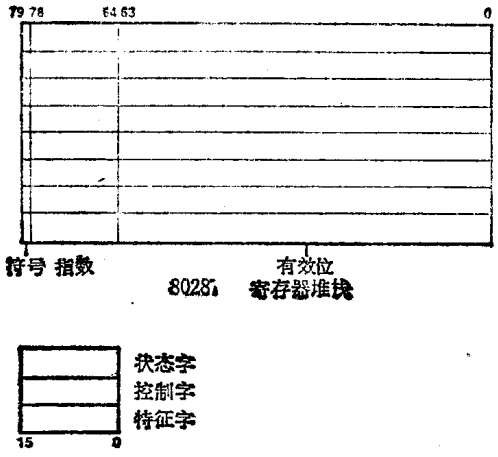


图 15.14　加上 80287 以后扩展的寄存器

2．扩展的数据类型

iAPX286 支持的所有数据类型，已经表示在图 15.13 中。这里我们把由 80287 支持的数据类型，重新表示在图 15.15 中。这些合宜的数据类型使所表示的数非常适合于应用，而且可以达到所要求的精度。

数据格式	范围	精度	最高字节
字整数	10^4	16 位	I_{15} … I_0　2 的补码
短整数	10^9	32 位	I_{31} … I_0　2 的补码
长整数	10^{19}	64 位	I_{63} … I_0　2 的补码
压缩的 BCD	10^{18}	18 数字	S – D_{17} D_{16} … D_1 D_0
短实数	$10^{\pm 38}$	24 位	S E_7 … E_0 F_1 · … F_{23}　F_0 隐含
长实数	$10^{\pm 308}$	53 位	S E_{10} … E_0 F_1 … F_{52}　F_0 隐含
临时实数	$10^{\pm 4932}$	64 位	S E_{14} … E_0 F_0 … F_{63}

整数：I

压缩的 BCD：$(-1)^S\ (D_{17}\cdots D_0)$

实数：$(-1)^S\ (2^{E-BIAS})(F_0 \cdot F_1\cdots)$

短实数 BIAS=127

长实数 BIAS=1023

临时实数 BIAS=16383

图 15.15　由 80287 支持的数据类型

(1) 整型数据

80287 支持 16 位、32 位和 64 位整型操作数. 这些整数使用 2 的补码格式表示负数，此时最高位为符号位.

16 位字整型格式，是 80286 和 80287 两个处理器共用的，以便确保在它们之间进行数据的快速传送. 32 位和 64 位整型格式，使在较大的整数之间的高精度运算得以进行. 64 位数据类型，使最大的整数可以表示为 9×10^{18}.

(2) 浮点数据类型

对于扩展精度的浮点运算，80287 支持了 32 位、64 位和 80 位实数. 这些标准化的浮点数据类型，包括一个符号位、一个指数字段和一个尾数字段.

32 位短实数值，提供 24 位精度和从 10^{-38} 到 10^{+38} 的数值范围；64 位长实数值，提供 53 位精度和从 10^{-308} 到 10^{+308} 的数值范围. 为了防止中间运算结果出现上溢或下溢，80287 采用了内部 80 位的数值格式，这种格式为连续运算提供了足够高的精度.

(3) 压缩的 BCD 数据类型

除了扩展的整型和浮点数据类型外，80287 还支持带符号的 18 位数字的压缩的 BCD 值. 压缩的 BCD 数据类型，具有从 0 到 10^{18} 的数值范围. 这些 10 进制数可以满足大多数实际的需要.

3. 扩展的寻址方式

80287 支持了三种操作数寻址方式. 第一种是隐含地把寄存器堆栈的栈顶元素、和可选择地把次堆栈顶元素作为操作数. 由于专用的数值指令，具有不需要完整的寻址灵活性的特点，所以可以使用这种方式，以节省存贮空间.

第二种寻址方式，是使一条指令能把任何其他的寄存器堆栈元素、和该堆栈的堆栈顶元素一起使用. 大部分双操作数指令使用这种寻址方式，以缩短和简化数值程序.

第三种方式是使用任何现有的 iAPX286 存贮器寻址方式，来引用存贮器操作数. 在存贮器和内部堆栈之间传送操作数的指令，使用这种技术.

4. 扩展的数值指令集

数值协处理器(或 NPX 的软件仿真)，对 iAPX 286 CPU 基本结构的指令集进行了扩充. 若在 80286 的指令序列中，放入 80287 的指令，系统就会按它们出现在指令流中的顺序，执行每条指令. 80287 和 80286 并行地进行处理，从而为数值计算提供了最大的吞吐量.

数值协处理器扩展了 CPU 的指令集，以支持高精度整数和浮点运算. 为了处理扩展的数据类型，扩展的指令集中，包含了算术运算、比较、超越函数、数据传送等指令. 80287 还含有一组有用的常数，以提高数值运算的速度.

(1) 算术指令

扩展的指令集不仅包含了 4 种算术操作(加、减、乘、除)，而且还包含了反向减和反向除指令. 算术函数包括求平方根、求模、求绝对值、取整、改变符号、化去指数和提取指数的指令.

(2) 比较指令

比较指令包括比较、检查和测试. 特殊形式的比较操作，由于可以直接把存贮器中的 2 进制整数和实数进行比较，故可以优化算法.

(3) 超越函数指令

这组指令完成所有原来很花费时间的常用的三角函数、反三角函数、双曲函数、反双曲函

数、对数函数和指数函数的核心计算。超越函数指令包括正切、反正切、2^X-1、$Y\cdot\log_2X$ 和 $Y\cdot\log_2(X+1)$。

(4) 数据传送指令

数据传送指令在寄存器之间、以及寄存器与存贮器之间传送操作数。这组指令包括装入、存放和交换指令。

(5) 常数指令

每一条常数指令，将一个通用的常数装入80287的寄存器。常数值具有64位实数精度，可以精确到19个10进制数字。用这些指令装入的常数包括0、1、π、$\log_210$、$\log_2e$、$\log_{10}2$ 和 $\log_e2$。

§15.4 iAPX 286 实地址方式

80286在实地址方式下，执行一个完全向上兼容的8086指令集的高级集。在实地址方式下，80286与8086和8088是软件目标代码相兼容的。实地址方式下的结构（寄存器和寻址方式），与前面关于iAPX286/10基本结构的介绍中的描述完全一致。

§15.4.1 存贮器容量

iAPX286系统在实地址方式下的物理存贮器，是一个彼此相连的，最大可到1,048,576字节的连续阵列。它们是由引脚 A_0 到 A_{19} 以及 $\overline{BHE}$ 寻址的。在实地址方式下，A_{20} 到 A_{23} 被忽略。

§15.4.2 存贮器寻址

在实地址方式下，处理器直接产生20位物理地址，这个地址实际上是由一个20位的段基地址和一个16位的偏移量组成。图15.16形象地表示了20位物理地基的形成过程，这个过

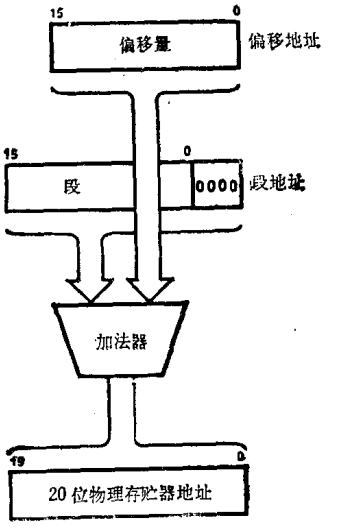


图15.16 iAPX286实地址方式地址计算

程是和 8086 的地址产生过程一样的。

指示器的选择子分量，被用作 20 位段基地址的高 16 位，而该基地址的低 4 位总是 0，因此段基地址总是从 16 字节的倍数开始。在实地址方式中，全部段的长度的上限，都是 64K 字节，并且这些段都可以被读、写或执行。若某个数据操作数、或指令，试图把一个段的首尾相连接，例如，一个字的低字节的偏移量是 0FFFFH，而它的高字节的偏移量是 0000H，则处理器将产生一个异常或中断。在实地址方式中，若一个段中的信息没有用完全部的 64K 字节，则该段的没有使用的部分，可以被其他的段所重叠，以便减少所需要的存贮器。

§15.4.3 保留的存贮器单元

80286 在实地址方式下，保留的两个存贮器区域如图 15.17 所示，即系统初始化区域和中断表区域。从地址 0FFFF0H 到 0FFFFFH 的单元是为系统初始化而保留的。初始化从地址为 0FFFF0H 的单元开始执行。从地址 00000H 到 003FFH 的单元是为中断向量保留的。

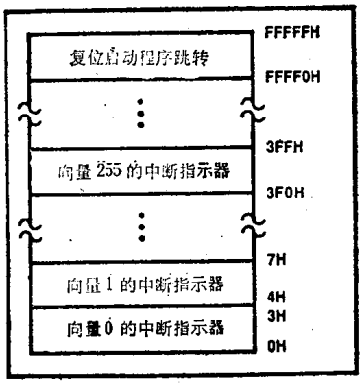


图 15.17 iAPX286 实地址方式保留的单元

§15.4.4 中　　断

表 15.14 所列出的，是为指出一个寻址错误的异常和中断所保留的中断向量。异常将使 CPU 停留在执行引起异常的指令之前的状态上 (PUSH、POP、PUSHA 或 POPA 除外)。

表 15.14 实地址方式寻址中断

功　能	中断数	相　关　的　指　令	返回指令前的地址吗?
中断表限太小异常	8	INT 向量不在表限内	返　回
协处理器段越出中断	9	存贮器操作数要超出偏移量 0FFFFH 的 ESC	不返回
段越出异常	13	用偏移量 0FFFFH 进行存贮器引用，或试图越过段尾执行	返　回

§15.4.5 保护方式初始化

为80286进入保护方式作准备，LIDT指令用来把中断描述子表的24位基地址和16位限装入中断描述子表的基地址和限寄存器。这条指令也可以在实地址方式下，为中断向量表建立一个基地址和限。在RESET以后，中断表基地址被初始化为000000H，并且把它的容量设置为03FFH。这些值是与iAPX86、88软件相兼容的。LIDT只能在为保护方式作准备时才可执行。

§15.4.6 停　机

当检测到一个严重错误时，就会发生停机，从而能防止CPU进一步处理下面的指令。停机和暂停，是通过一个暂停总线操作而被外部通知的。它们可以以A_1为高电平对应暂停和A_1为低电平对应停机来区别。在实地址方式下，停机在下列两种情况下发生：

(1) 异常8或13发生，并且IDT的限不包含该中断向量；

(2) CALL、INT或POP指令，在SP不是偶数时试图首尾连接堆栈段。

若IDT的限至少是000FH，并且SP大于0005H，则一个NMI输入可以把CPU带出停机，否则停机状态只能通过RESET来终止。

§15.5 iAPX 286保护虚地址方式

当保护虚地址方式下，iAPX286把实地址方式的能力和存贮器管理、对虚拟存贮器的支持以及对地址空间的保护集为一体，从而使iAPX286能可靠地支持多用户系统。这样，就把16位微处理机的性能提高到了一个新的水平，而它的成本却远较发展和制造其他的系统为低。

作为iAPX286系统CPU的80286，在保护虚地址方式中，执行一个与8086指令集完全向上兼容的高级集。保护虚地址方式，还提供了存贮器管理机构和保护机构以及有关的指令。

80286使用LMSW(Load Machine Status Word)指令，来置位机器状态字中的PE(Protection Enable)位，从而使处理器本身从实地址方式，进入保护虚地址方式。保护方式提供扩展了的物理和虚拟地址空间、存贮器保护机构和新的操作，用来支持操作系统和虚拟存贮器。

前面在iAPX286/10基本结构部分中描述的所有寄存器、指令和寻址方式，在iAPX286保护虚地址方式下均保持不变。iAPX86、88、186的程序和80286的实地址方式下的程序，都可以在保护虚地址方式下运行，但是，在这种情况下为段选择子所预置的常数是不同的。

下面，我们将着重从对保护虚地址方式得以实行的硬件支持、和它们的有关数据结构出发，来介绍iAPX286的保护虚地址方式。

§15.5.1 存贮器容量

在保护虚地址方式中，iAPX286具有16兆字节的存贮器寻址能力，通过集成在片内的保

护虚地址机构，iAPX286 对每个任务提供了最大可达 1000 兆字节的虚拟存贮器空间。如同在实地址方式中一样，程序并不涉及具体的物理地址。不同的是，段寄存器现在所指的，是虚地址空间的 16000 个 64K 字节段中的某一个。有效地址，指的是段内所求操作数的偏移量。虚地址到实地址之间的转换，是由 iAPX286 片内的存贮器管理部件自动完成的。这样既节省了管理外部存贮器所需要的专用指令或子程序，又提供了较高的系统吞吐率。

iAPX286 保护方式下，虚地址空间的分段，与实地址方式下地址空间的分段，是类似的。这种设计的兼容性，简化了程序从实地址方式到保护虚地址方式转化的过程。

保护方式中使用的地址，是实地址方式下实地址的自然扩展，从而为应用程序提供了兼容性。

虚拟地址空间，比物理地址空间大，这是由 80286 的保护机构作为支柱的。每当一条指令引用任何不能映射到物理存贮器的地址时，处理器就会引起一个可重新启动的指令异常，在进行必要的调进调出以后，指令就能正确地执行。

§15.5.2 存贮器寻址

和实地址方式一样，保护方式也使用了 32 位指示器，它包含着一个 16 位的选择子分量，和一个 16 位的偏移量分量，但是选择子的内容，不再是一个实际地址的高 16 位，而是进入某一个存贮器常驻表的变址值。所要求的段的 24 位基地址，要从在存贮器中的表中取得。16位偏移量用来加到段基地址，来形成物理地址。图 15.18 表示了 24 位物理地址的形成过程。一旦一个选择子被装入了一个段寄存器，CPU 就会自动地引用这些表，来取得段基地址和其他信息。所有装入一个段寄存器的 iAPX286 指令，不需要附加的软件，就能引用存贮器。包含 8 字节值的存贮器表，叫做描述子。iAPX286 使用这种被称为描述子的数据结构，在存贮器

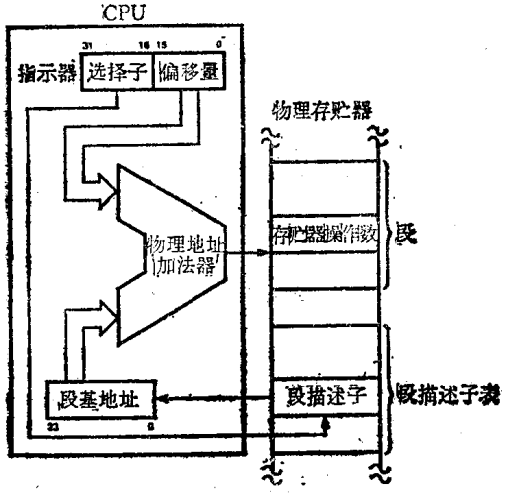


图 15.18 保护虚地址方式存贮器寻址

寻址的过程中提供了一个间接层。有了这个间接层，80286 的存贮器管理机构和保护机构就有了活动的舞台。例如操作系统利用这种间接层，就能够把某一个段放在实存中的任意位置，而不需要调整程序的地址常数，因此，用这种方法，既保持了实地址方式中的基本寻址过程，又使得 iAPX286 省去了为使用虚存而重新编写应用程序的需要。

一、选择子

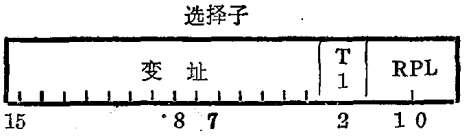
一个保护方式下的选择子，有3个字段：

(1) 描述子入口变址，它可以寻址8K个描述子；

(2) 局部或全局描述子表指示器(TI)，用来确定在局部描述子表中、还是在全局描述子表中、去寻找所要找的描述子；

(3) 选择子特权，表示所请求的特权层。

图15.19表示了选择子的这3个字段。这些字段用来选择两个存贮器描述子表中的一个，以便选择合适的表项，并使保护机构能对选择子特权属性，进行高速测试(参见后面关于特权的讨论)。



位	名　　称	功　　能
1—0	REQUESTED PRIVILEGE LEVEL(RPL)	表示所需要的选择子特权层
2	TABLE INDICATOR(TI)	TI=0 使用全局描述子表(GDT) TI=1 使用局部描述子表(LDT)
15—3	变址	在表中的选择子项

图 15.19　选择子字段

二、描述子

描述子定义了存贮器的用途，特殊类型的描述子，则定义了控制转移和任务转换的新的功能。80286有关于代码、堆栈和数据段的段描述子、和关于特殊系统数据段和控制转移操作的系统控制描述子。在多处理器系统中，描述子的访问，是作为封锁总线操作来执行的，这样可以确保描述子的完整性。

三、代码段和数据段描述子

代码段和数据段描述子，除了段基地址外，还包含了其他的段属性，主要有：

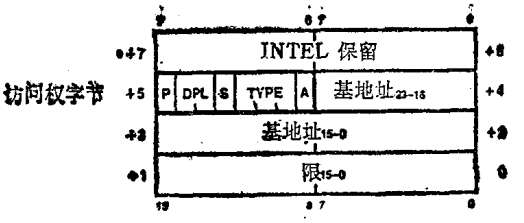
(1) 段长的限(1到64K字节)；

(2) 访问权(只读、读/写、只执行和执行/读)；

(3) 在存贮器中是否存在(对虚拟存贮器系统而言)；

(4) 描述子特权层。任何违反由段描述子规定的段属性的对该段的使用，将会引起一个异常或中断。图15.20是代码和数据段描述子的格式，以及该种描述子中访问权字节中各字段的功能。

图15.21(a)是一个描述子的例子，为了分析是一个什么类型的描述子，可以先把访问权



位	名　称	功　能	
7	Present(P)	P=1 该段在物理存贮器中存在 P=0 该段不在物理存贮器中，基地址和限没有用.	
6—5	Descriptor Privilege Level (DPL)	在特权测试中所使用的段的特权属性.	
4	Segment Descriptor(S)	S=1 代码段或数据段描述子 S=0 非段描述子	
TYPE字段定义 3	Executable(E)	E=0 数据段描述子类型是：	数据段
2	Expansion(ED) Direction	ED=0 向上生长段，偏移量必须≤限 ED=1 向下生长段，偏移量必须＞限.	数据段
1	Writeable(W)	W=0 数据段不能被写入 W=1 数据段可以被写入.	数据段
3	Executable(E)	E=1 代码段描述子类型是：	代码段
2	Conforming(C)	C=0 当 CPL＞DPL 时，代码段只能被执行.	代码段
1	Readable(R)	R=0 代码段不能被读 R=1 代码段可以被读	代码段
0	Accessed(A)	A=0 该段还没有被访问过. A=1 段选择子已被装入段寄存器或已被选择子测试指令使用过.	

图 15.20　代码和数据段描述子

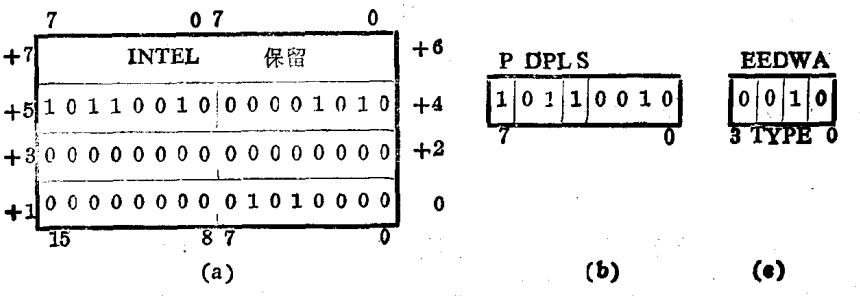


图 15.21　一个数据段描述子

字节取出来(图 15.21(b))，先看 S 字段，这里 S = 1，说明是一个代码或数据段描述子，这样就可以知道 S 字段后面的 3 位是 TYPE 字段，由于 TYPE 字段中的最高位，即第 3 位是 0，所以该描述子是一个数据段描述子. 于是我们可以用图 15.21(c)，来表示访问权字节的低 4 位. 从图中可以知道，ED = 0，表示这是一个向上生长段，而 W = 1，表示该段是一个可写入的数据段. 进一步的分析使我们了解到该段在存贮器中的基地址是 0A0000H，而该段的限为

0050H．由于P=1，该段存在于存贮器中，A=0，表示该段没有被访问过．我们进一步假设要访问的操作数的偏移量为0030H，由于0030H≤0050H，所以满足ED=0所规定的限制．这样，操作数的实际地址就是：

$$\begin{array}{r} 0A0000H \\ +\quad 0030H \\ \hline 0A0030H \end{array}$$

由于DPL字段的值为1，所以该数据段是处于特权层1中．

让我们再来看一个描述子的例子，它的结构如图15.22(a)所示．分析的过程同上个例子一样，先把访问权字节取出来，如图15.22(b)所示．由于S=1，所以是一个代码数据段描述

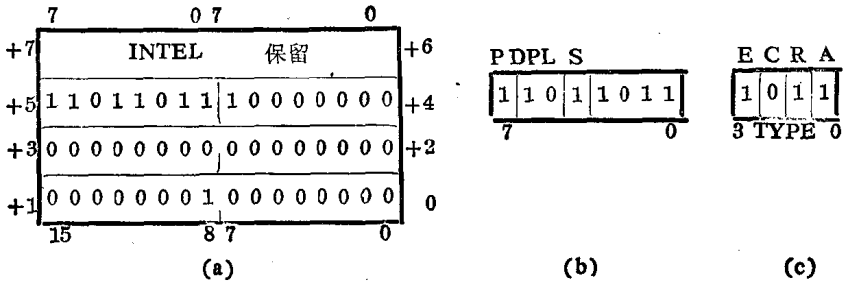


图15.22 一个代码段描述子

子，因而可以把访问权字节的低4位用图15.22(c)来表示．由于TYPE字段的最高位，即第3位E=1，所以TYPE字段中的后2位可以分别用C、R字段来表示．由于C=0，所以该代码段在CPL≥DPL，即当CPL≥2时，该代码段只能被执行（CPL的含义在以后介绍）．由于R=1，所以该代码段是可读的．回过头来再看图(b)，由于P=1，所以该代码段存在于存贮器中，又由于A=1，说明该段已经被访问过．从描述符中的基地址字段和限字段中可以得知，该代码段的基地址是800000H，而它的限是0100H．若我们进一步假设当前要寻址的指令的偏移量为0040，则该条指令在存贮器中的绝对地址为：

$$\begin{array}{r} 800000H \\ +\quad 0040H \\ \hline 800040H \end{array}$$

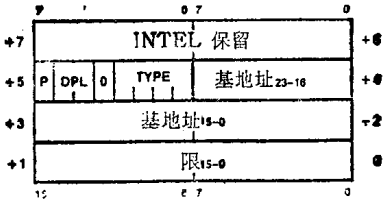
由于DPL=2，所以该代码段是处于特权层2中．

四、系统控制描述子

除了代码段和数据段描述子外，保护方式下的80286，还定义了系统控制描述子．这些描述子在保护环境下，定义特殊的系统数据段和控制转移机构．特殊系统数据段描述子，定义了含有描述子表的段（局部描述子表描述子），和含有任务执行状态的段（任务状态段描述子）．这二种描述子的格式和字段定义如图15.23所示．

这类描述子包含了一个24位的段基地址、和一个16位的限，它的访问权字节，则规定了它的状态和所处的特权层．若P=1，则该描述子的内容是有效的，并且该段是在物理存贮器中．若P=0，则该段是无效的．DPL字段，仅在任务状态段描述子中使用，它指出可以使用该描述子的特权层．由于局部描述子表描述子只能被一条专用的特权指令使用，所以在局部

描述子表描述子中没有使用 DPL 字段。访问权字节的第 4 位，即 S 字段为 0，表示是一个系统控制描述子。



字 段 名	值	说 明
TYPE	1	可用任务状态段
	2	局部描述子表描述子
	3	忙任务状态段
P	0	描述子内容无效
	1	描述子内容有效
DPL	0—3	描述子特权层
基 地 址	24位数	在实际存贮器中的特殊系统数据段的基地址
限	16位数	在该段中的最后一个字节的偏移量

图 15.23 特殊系统数据段描述子

下面用两个例子来说明这类描述子。图 15.24 是一个局部描述子表描述子的例子。其中

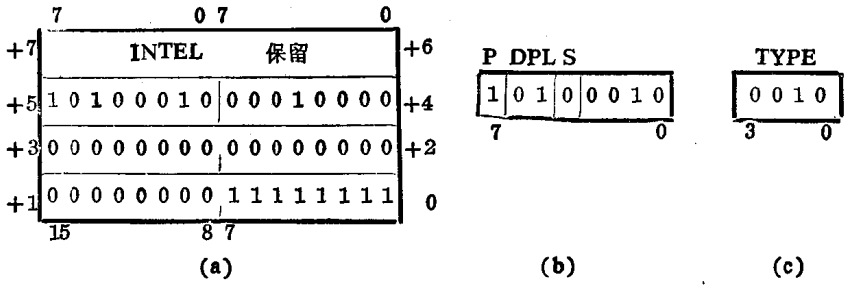


图 15.24 一个局部描述子表描述子

(a)是描述子，(b)是该描述子的访问权字节，由于 S 字段为 0，所以图(b)的低 4 位，可以用图(c)来表示。由于 TYPE = 2，所以就确定了该描述子是一个局部描述子表描述子，因而该访问权字节中的 DPL 字段，是没有用的。由于 P = 1，所以该描述子的内容是有效的（若 P = 0，则说明该描述子所指的局部描述子表还在外存，在把该局部描述子表调入内存以后，保护机构会把它置为 1）。于是，我们可以知道从绝对地址 100000H 开始，是一个局部描述子表，该表的长度是 256 个字节，也就是说，该局部描述子表中可以放入多至 32 个局部描述子。

图 15.25，是一个任务状态段描述子。其中(a)是描述子，(b)是该描述子的访问权字节。由于 S = 0，所以该描述子是一个系统控制描述子，因而图(b)的低 4 位，可以用图(c)来表示。由于 TYPE = 3，所以该描述子是一个任务状态段描述子，从图(a)中可知，该任务状态段的基地址为 200000H，该段的限为 0030H，而该描述子的特权层 DPL = 2。

系统控制描述子中，除了上述的特殊系统数据段描述子外，还有一类控制转移描述子。这类描述子，通常称为门描述子，简称为门。与存贮器的段描述子不一样，门描述子定义了一个

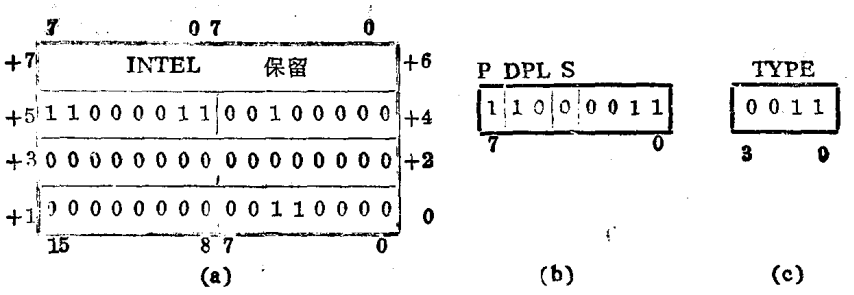
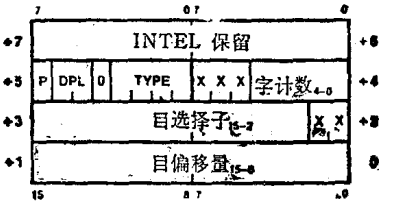


图 15.25 一个任务状态段描述子

代码段的保护入口，它们是：调用门、任务门、中断门和自陷门。顾名思义，门就是一种关卡，80286 设置门这样一种机构，就在控制转移的源和目之间，建立了一道关卡，这样 CPU 就能在这道关卡上，自动地执行保护检查和控制目的入口。调用门可以用来改变特权层，任务门可以用来实现一个任务转换，而中断门和自陷门则用来指定中断服务程序。使用中断门将禁止中断(复位 IF)，而自陷门却不这样做。图 15.26 是门描述子的数据格式和各字段的定义。该种描述子，包含一个目的指示器，它指向目标段的描述子，以及在该目标段中入口点的偏移量。中断门、自陷门和调用门中的目选择子，必须引用一个代码段描述子(不是一个代码段!)。这些



字 段 名	值	说 明
TYPE	4 5 6 7	——调用门 ——任务门 ——中断门 ——自陷门
P	0 1	——描述子内容是无效的 ——描述子内容是有效的
DPL	0—3	——描述子特权层
字 计 数	0—31	要从调用程序的堆栈中拷贝到被调用的子程序堆栈中去的字数，这个字段只有调用门使用
目选择子	16位选择子	目的代码段(调用、中断或自陷门)的选择子 目的任务状态段(任务门)的选择子.
目偏移量	16位偏移量	在目的代码段中的入口点

图 15.26 门描述子

门描述子包含着入口点的偏移量，这样可以防止一个程序建立和使用非法的入口点。任务门只能引用一个任务状态段。由于任务门调用一个任务转换，所以任务门中的目偏移量没有用。

当某个门被使用时，若其中的目选择子不引用正确的描述子类型，就会引起异常 13。字

计数字段在调用门描述子中，用来表示参数的个数(0—31 个字)，这些参数在一个改变特权层的控制转移中，要被从调用程序的堆栈中，自动地拷贝到被调用程序的堆栈中去。其他的门描述子都不使用字计数字段。

访问权字节的格式，对所有的门描述子都是一样的。P = 1，表示该门的内容是有效的；P = 0，则表示该门的内容无效。若对无效的门引用，就会引起异常 11。DPL 是描述子特权层，它在该描述子被一个任务使用时指定。第 4 位必须等于 0，以表示是一个系统控制描述子，类型字段则指定了描述子的类型。

图 15.27 是一个调用门的例子，其中(a)是描述子，(b)是访问权字节，由于 S = 0，所以它的低 4 位可以用(c)来表示。由于TYPE = 4，所以该描述子是一个调用门，因而它的字计数字段可以用图(d)来表示。由于 P = 1，这个门的内容是有效的，该门描述子的特权层为 2(DPL = 2)，

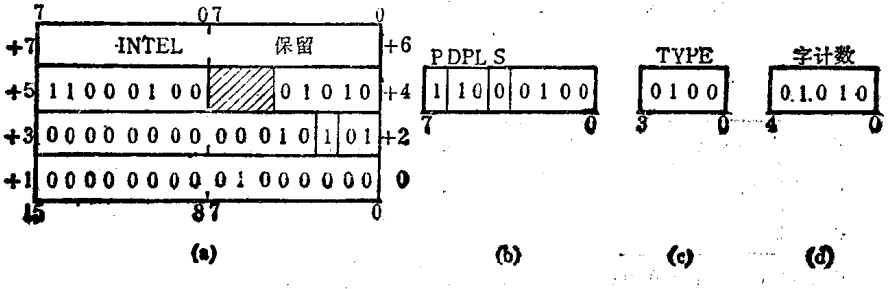


图 15.27 一个调用门描述子

由于字计数字段值为 10，所以在调用过程中，要通过堆栈传递的参数个数为 10。通过该门描述子所要调用的代码段描述子，由该门中的选择子到局部描述子表(选择子中 TI = 1) 中的第 2 项取得，该选择子所请求的特权层是 1(RPL = 1)。被调用的程序的入口偏移量，是该门描述子中所指出的 0040H。

图 15.28 是一个任务门的例子，其中(a)是任务门，(b)是该门中的访问权字节，而(c)则是

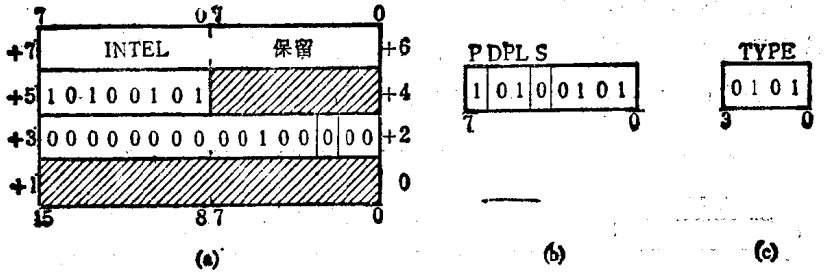


图 15.28 一个任务门描述子

该访问权字节中的 TYPE 字段。由于 TYPE = 5，所以它是一个任务门描述子。它表示要在全局描述子表中(选择子的 TI 字段为 0)的第 4 项，取得任务状态段描述子。由于任务门只能引用一个任务状态段，所以该任务门中的偏移量字段是不用的。

图 15.29 是一个中断门的例子，其中(a)是中断门，(b)是该中断门的访问权字节，(c)是访问权字节中的 TYPE 字段。这个例子中的中断门，要引用在局部描述子表的(选择子的TI = 1)第 32 项中的代码段描述子，而中断处理程序在该代码段中的入口的偏移量是 0028H。

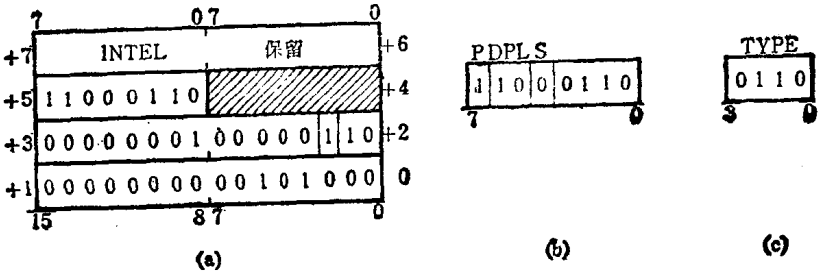


图 15.29　一个中断门描述子

图 15.30 是一个自陷门的例子，其中(a)是自陷门，(b)是该自陷门中的访问权字节，而(c)则是该访问权字节中的 TYPE 字段。这个例子中的自陷门，要访问一个在全局描述子表(选择子的 TI = 0) 的第 16 项中的代码段描述子，而自陷处理程序在该代码段中的入口点的偏移量是 0200H。

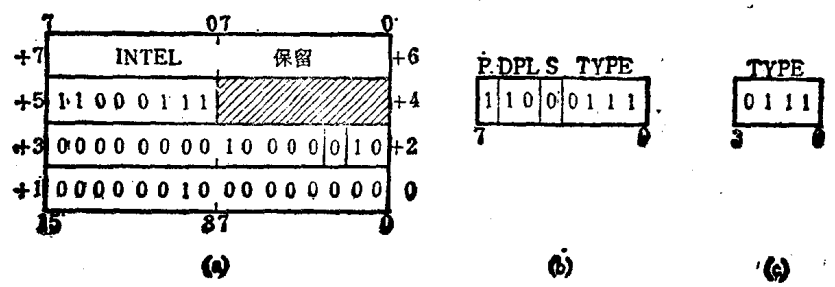


图 15.30　一个自陷门描述子

五、段描述子 CACHE 寄存器

段描述子 CACHE 寄存器，是实地址方式下的段寄存器的扩展，就是 80286 对段描述子数据结构的硬件支持。段寄存器 (CS、SS、DS、ES) 中的每一个都分配有一个段描述子 CACHE 寄存器。这些段描述子CACHE寄存器，对程序员来说是透明的，每当一个选择子被装入一个段寄存器时，由该选择子所决定的段描述子，也就被自动地装入到相应的 CACHE 寄存器中去。图 15.31 表示了描述子 CACHE 寄存器的结构、和与段寄存器的对应关系。显然，只有段描述

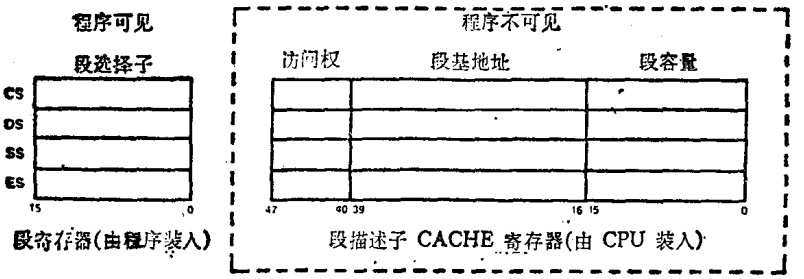


图 15.31　引描述子 CACHE 寄存器

子，才能被装入到段描述子 CACHE 寄存器中去。一旦某一个段描述子装入了相应的 CACHE 寄存器，则对存贮器中该段的访问，就再也不要去访问描述子表中的描述子，只要访问CACHE 寄存器中的内容就可以了。毫无疑问，访问 CACHE 寄存器所需要的时间，要比访问存贮器所

需要的时间少得多，同时，采用这种方法也有利于对集中的管理信息进行处理。80286 并不提供存放这些 CACHE 寄存器内容的指令，这些寄存器的内容，只有在一个段寄存器被装入时才会改变。

图 15.32 是一个说明段寄存器和段描述子 CACHE 寄存器配合，来共同确定存贮器中 4 个当前段的例子。图中装在 4 个段寄存器中的选择子，指明了 4 个相应的段描述子都在当前

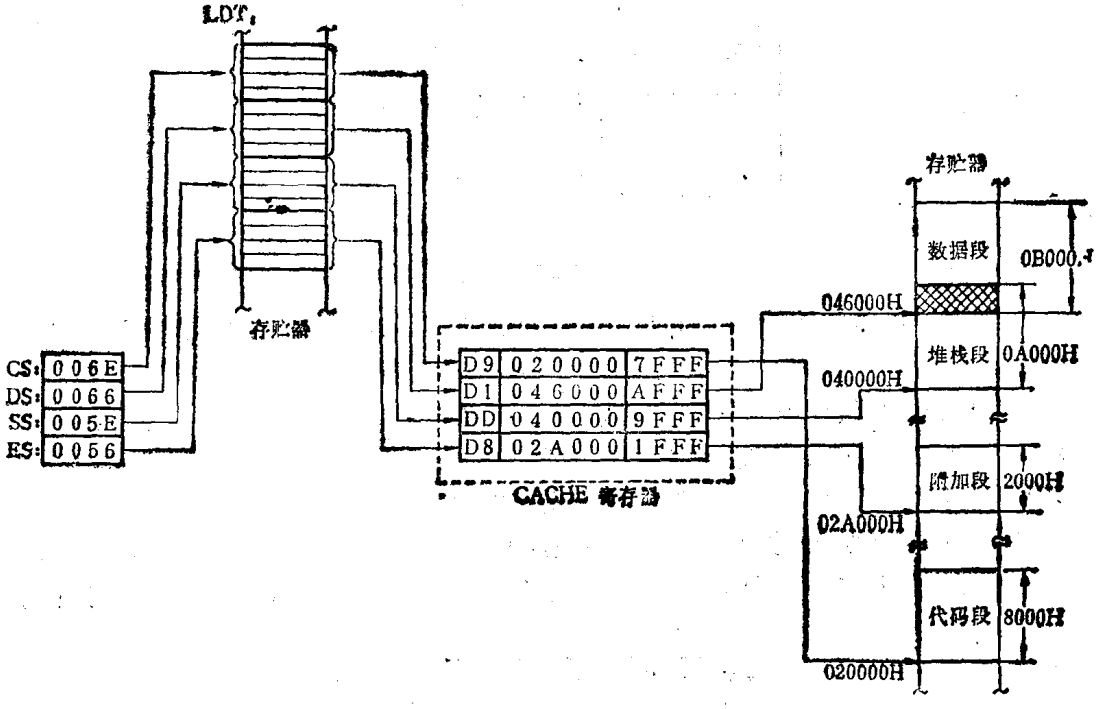


图 15.32　段寄存器和 CACHE 寄存器相配合，确定存贮器中的 4 个当前段

局部描述子表中，并且是在该表的第 10—13 项中。现在假定 4 个段描述子，已经分别被装入相应的 CACHE 寄存器，它们指定了各个当前段的基地址和段的容量。各个描述子中的访问权字节，给这些段规定了些什么限制，请读者思考。

六、局部和全局描述子表

描述子表，是一个最多可以有 8192(8K)个描述子的线性数组。iAPX286 用这种表，来形成操作系统、软件和虚地址寻址硬件之间的接口。前面已经提到过，选择子的最高 13 位值，是进入某一个描述子表的变址值，实际上，就是指进入全局描述子表或局部描述子表的变址值。全局描述子表和局部描述子表，包含了在任何时刻可以被某一任务访问的全部描述子。这两个表都有一个基地址寄存器，用来存放在物理存贮器中的描述子表的 24 位基地址，同样，它们也都有一个 16 位的限寄存器，这两个寄存器把对描述子表的访问，限止在如图 15.33 所表示的限内。这两组寄存器对在 80286 保护虚地址方式下访问描述子的操作，提供了硬件支持。其中局部描述子表的一组寄存器，对程序来说是透明的，局部描述子表选择子寄存器，用来存放寻找当前局部描述子表描述子的选择子。任何时候，若试图引用在表限外面的描述子，就会引起一个可重新启动的异常(13)。

全局描述子表 GDT(Global Descriptor Table)，包含着可供所有任务使用的描述子。GDT

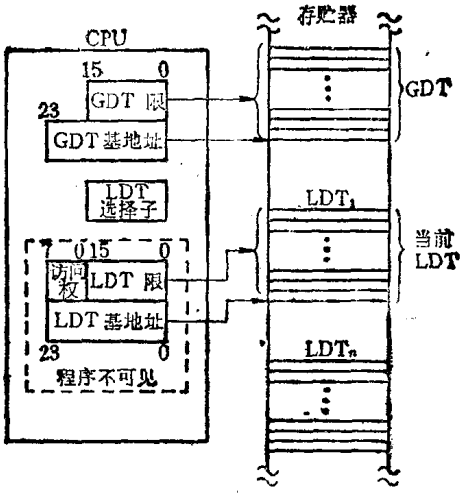


图 15.33 局部和全局描述子表定义

包含除了中断和自陷门描述子之外的所有类型的描述子。局部描述子表 LDT(Local Descriptor Table)，包含着一个作业所私有的描述子。每个任务可以有它自己私有的 LDT。与 GDT 不同，LDT 只能包含段、任务门和调用门描述子。若在某一时刻，某个段的描述子在两个表中都不存在，则此时该段就不能被一个任务所访问。

LGDT 和 LLDT 指令，用来把全局和局部描述子表的基地址和限装入 GDT 和 LDT 寄存器。LGDT 和 LLDT 是受保护的，只有在特权层 0 操作的程序才能执行它们。LGDT 指令把在存贮器中的如图 15.34 所示的含有全局描述子表的 16 位限和 24 位基地址的 6 字节字段，

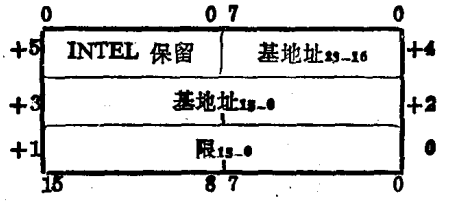


图 15.34 装入全局描述子表寄存器和中断描述子表寄存器的数据类型

装入 GDT 寄存器。LLDT 指令则有些不同，它是把在一个 16 位寄存器中或在一个 16 位存贮器操作数中的选择子，装入 LDT 选择子寄存器，然后处理器用在该寄存器中的选择子，在全局描述子表中引用一个如图 15.23 所示的、含有一个 LDT 的基地址和限的局部描述子表描述子。和前面叙述的 CACHE 寄存器的装入一样，LDT 寄存器的装入，也是自动进行的。

七、中断描述子表

保护方式的 80286，还具有第三种描述子表，即中断描述子表 IDT(Interrupt Descriptor Table)。它用来定义多至 256 种中断。在中断描述子表中，只能包含任务门、中断门和自陷门。80286 为这种数据结构提供了一个 24 位的基地址寄存器、和一个 16 位的限寄存器，这些寄存器同中断描述子表的关系，表示在图 15.35 中。受保护的指令 LIDT，用来把和 LGDT 指令所用的相同格式的 6 字节值(见图 15.34) 装入这些寄存器，这 6 个字节在存贮器中的起始地址，由 LIDT 指令给出。

引用 IDT 项目的操作，是由 INT 指令、外部中断向量或异常来进行的。由于 Intel 保留了

32 种中断，所以 IDT 必须至少要有 256 个字节，来为所有保留的中断分配空间(32×8=256)。在保护方式下，寻找中断处理程序的过程和实地址方式下也有所不同。在保护方式下的 80286，接到一个中断向量后，就自动地引用中断描述子表寄存器，通过是否越限检查之后，就用中断描述子表基地址寄存器中的值、和中断向量共同确定该种中断所对应的中断门在表中的位置，通过访问权检查之后，就引用该中断门。利用该中断门中的选择子，到全局描述子表中寻找中断处理程序所在代码段的描述子，找到以后，经过访问权检查，就引用该代码段描述子。利用代码段描述子提供的代码段的基地址加上在中断门中提供的中断处理程序的入口偏移量，就找到了对应于 80286 所接收到的中断向量的处理程序的入口，从而就能开始执行中断服务。

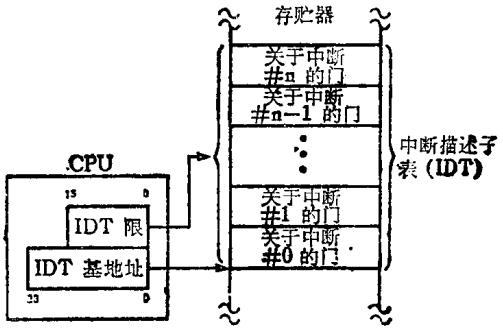


图 15.35 中断描述子表定义

至此，我们已经了解到，为了支持保护虚地址方式的寻址，iAPX286 一共提供了三种描述子表，它们的关系如图 15.36 所示。其中 GDT 和 IDT 是各个任务共享的，而 LDT 则由各相应的任务私用。图 15.37 是某一时刻三种描述子表和它们相应的寄存器的关系。其中全局描

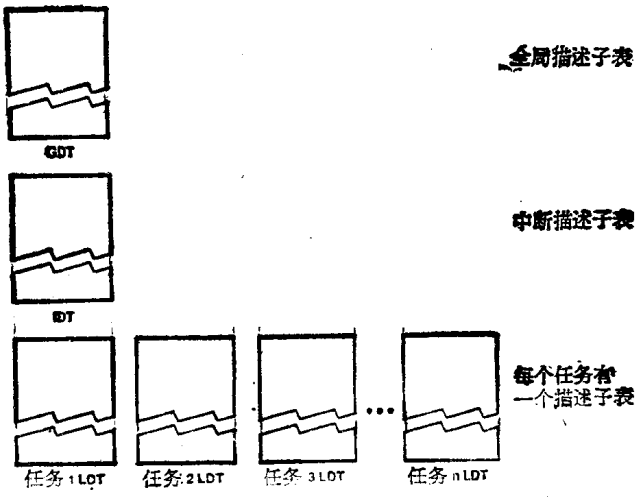


图 15.36 iAPX286 的描述子表

述子表的基地址为 300000H，表的限为 7FFFH，即表示该表中可放 4K 个描述子。中断描述子表的基地址为 200000H，表限为 07FF，表示可存放 256 个描述子。值为 0021H 的局部描述子表选择子，是由特权指令 LLDT 装入的，装入以后，它引用在全局描述子表的第 4 项中的局部描述子表描述子，并把它的内容装入局部描述子表寄存器。该局部描述子表的基地址是 100000H，表的限为 00FFH，即表示该表可以放 32 个描述子。

八、存贮器寻址小结

从上面的介绍中，我们看到 iAPX286 是用一种所谓选择子和描述子的数据结构，来实现保护虚地址方式下的寻址的。这样做的目的，不仅是为了能寻址 16 兆的实存地址空间，更重要的是给操作系统和 80286 的保护虚地址机构，提供了一个宽广的活动舞台，从而使 iAPX

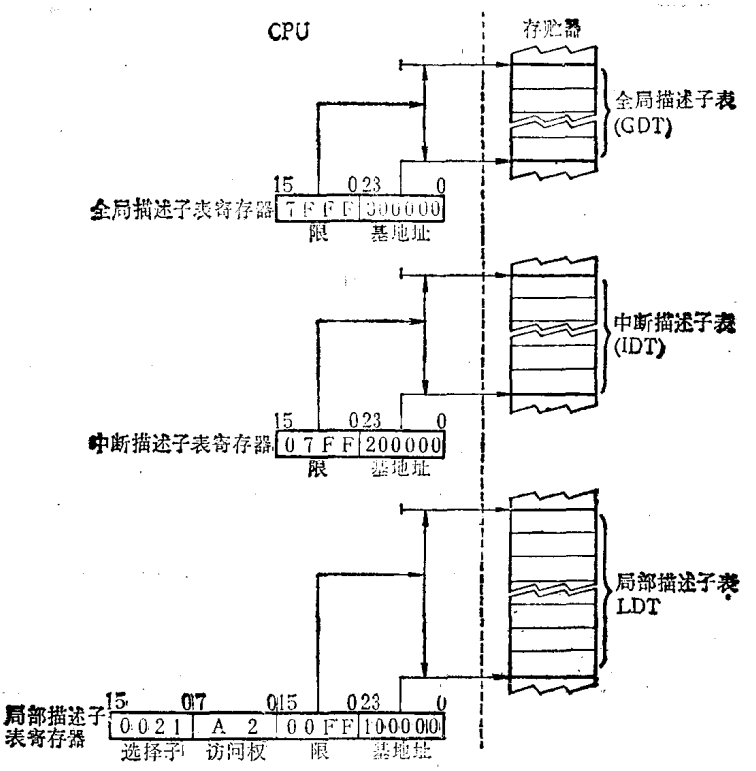


图 15.37 描述子表寄存器和描述子表

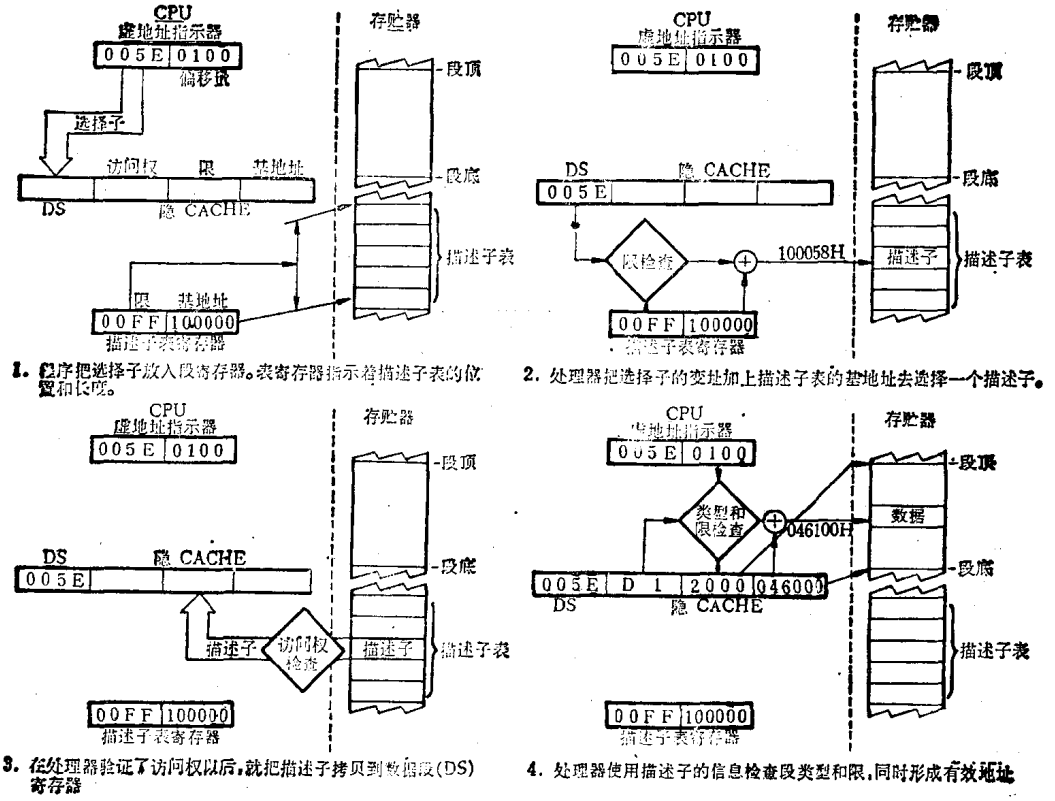


图 15.38 寻址一个操作数的例子

286 系统的各种性能得以实现。为了避免因引用这些数据结构，而频繁地访问存贮器，80286 提供了相应的硬件寄存器，从而使因访问这些数据结构而产生的对处理器运行速度的影响减至最小。最后，我们以一个例子，来结束关于 iAPX286 在保护虚地址方式下寻址的讨论。为了能简洁地叙述，我们在这里忽略了一些非本质的细节。

假定，CPU 要根据一个如图 15.38(1) 中所示的虚地址指示器，到存贮器中去取一个操作数。为执行这样一个操作，处理器先把指示器中的选择子，装入 DS 寄存器，然后如图 (2) 所示，根据选择子的 TI 字段(TI = 1)，把选择子的变址值 000BH，乘以 8H 以后的值 0058H、与在局部描述子表限寄存器中的值 00FFH 进行比较，得知该变址值没有超过局部描述子表的限，因此可以进行描述子表的访问。把进入描述子表的变址值 0058H、与局部描述子表的基地址值 100000H 相加，得到所要访问的描述子的绝对地址 100058H。随后，如图 (3) 所示，CPU 的保护机构，就对该描述子的访问权进行检查，假定对该描述子的访问是合法的(具体过程在后面关于特权和保护的讨论中介绍)，这样，CPU 就自动地把该描述子中的所要访问的数据段的基地址、限和访问权字节装入相应于 DS 寄存器的 CACHE 寄存器。最后，如图 (4) 所示，用已装入 CACHE 寄存器的访问权字节中的类型字段所规定的内容，来对 CPU 的数据段引用请求进行检查，同时，还要对指示器中的偏移量进行检查，以便确定该偏移量是否超出了为数据段所规定的限。显然这个偏移量没有超出数据段的限，于是，就可以把偏移量和数据段的基地址相加，即：

$$\begin{array}{r} 046000H \\ +\quad 0100H \\ \hline 046100H \end{array}$$

这样，就最终取得了存有所要的操作数的存贮器单元的 24 位物理地址。

15.5.3 特　　权

80286 为了支持操作系统程序和任务程序的分离、和任务与任务之间的分离，提供了一个具有 4 个等级的特权系统。利用这种特权系统，保护机构就可以在由描述子提供的中间层上实

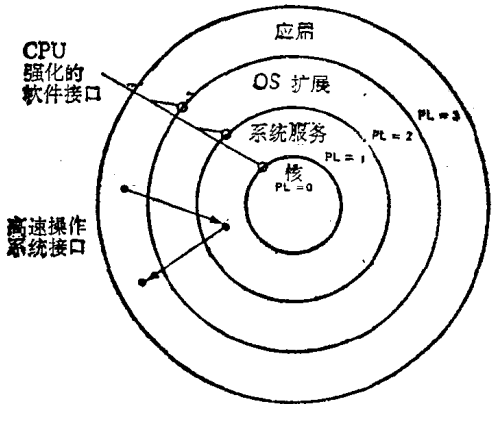


图 15.39　特权层结构

现保护操作。例如，使用特权可以控制特权指令的使用和在一个任务中对描述子（以及与它们相关的段)的访问。80286 支持的 4 层特权如图 15.39 所示，这是通常在小型计算机中使用

的用户/监督方式的一种扩充形式。特权层的标号为0到3，其中0层是最高特权层，3层是最低特权层。特权层在一个任务中，用来提供保护(任务之间的保护，是由为每个任务提供一个私有的LDT来进行的)。操作系统程序、中断处理程序和其他系统软件，可以根据需要分别处于不同的特权层次中，而得到相应的保护，因此它们可以和应用程序在同一个任务的地址空间中共处，而不发生越权操作。一个任务，对每一个特权层都有一个独立的堆栈。任务、描述子和选择子都有一个特权属性，这种属性决定了描述子是否可以被使用。任务特权影响指令和描述子的使用，描述子和选择子的特权则仅影响对该描述子的访问。

一、任务特权

一个任务，总是在4个特权层中的一个中执行。任务在特定时刻的特权层，称为当前特权层CPL (Current Privilege Level)。它是由CS寄存器的最低2位来规定的。当在一个代码段中执行时，CPL不能被改变。一个任务的CPL，只有在通过门描述子把控制转移到一个新的代码段时(见控制转移部分)，才能改变。

当任务通过一个任务转换操作启动时，该任务就在由代码段寄存器所指定的CPL值所规定的特权层上执行。在0层执行的任务，可以访问在GDT和该任务的LDT中定义的所有数据段，并且被认为是处于最高特权层，而在第3层执行的任务对数据的访问，将受到最大的限制，并且被认为是处于最低特权层。

图15.40，是某一时刻CS寄存器的内容，由于CS寄存器内容的最低2位的值是2，所以在此特定时刻的任务特权层，即当前特权层CPL = 2。

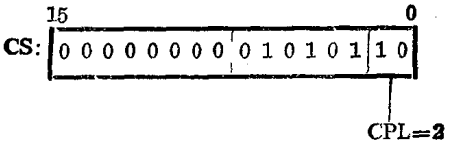


图15.40 CS的最低2位决定CPL

二、描述子特权

描述子特权，是由描述子的访问权字节中的描述子特权层DPL规定的。DPL规定了可以访问该描述子的任务的最低任务特权层，也就是当前特权层CPL值的最高界限。换句话说，只有满足条件CPL≤DPL(数值)时，当前的任务才可以访问该描述子。具有DPL = 0的描述子，是受到最多保护的，只有在特权层0执行的任务，才可以访问它们。具有DPL = 3的描述子是受到最少保护的(即具有最少的访问限制)，因为CPL = 0、1、2或3的任务，都可以访问它们。除了LDT描述子外(该描述子的DPL字段没有意义)，这个规则适用于所有的描述子。

图15.41是使用这个规则的例子。CS寄存器的内容，表示要访问在当前局部描述子表的

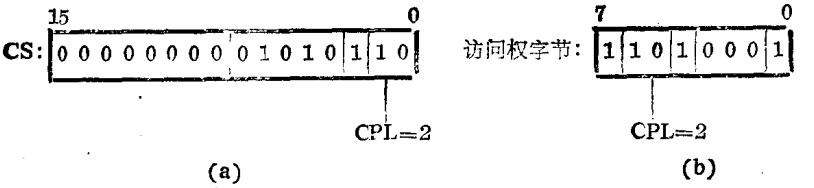


图15.41 DPL限制了对描述子的访问

第 10 项中的描述子，(a)表示了 CS 寄存器；(b)表示了所要访问的描述子的访问权字节。由于 CPL = DPL，所以对该描述子的访问是允许的。若 (b) 中的 DPL = 0、1，则这种访问是不允许的，然而，当 DPL = 3 时，这种访问照样能进行。

三、选择子特权

选择子特权，是由在一个选择子的最低 2 位的请求优先权字段 RPL (Requested Privilege Level) 规定的。选择子 RPL，为该选择子的使用而建立比当前特权层更低的特权层。这一特权层称为任务的有效特权层 EPL (Effective Privilege Level)。RPL 只能减少任务使用这个选择子访问数据的范围。一个任务的有效特权，是 RPL 和 CPL 中的数值较大者，即 EPL = max (RPL, CPL)。具有 RPL = 0 的选择子，在使用上没有附加的限制，而具有 RPL = 3 的选择子，则不管任务的 CPL 是什么，只能引用在特权层 3 的段。RPL 一般用来确保传送到一个具有更高特权的过程的指示器参数，不能使用具有比调用程序更高特权层的数据(详见指示器测试指令)。

图 15.42，是使用选择子特权的例子。其中(a)是当前 CS 寄存器的值；(b)是当前 ES 寄存器的值，假定现在要通过 ES 寄存器，引用一个数据段描述子，而在该数据段中存放一个数据。

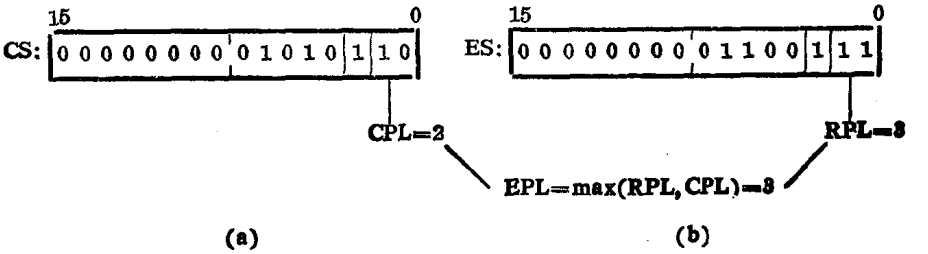


图 15.42 选择子特权的作用

此时，CPL = 2，而 RPL = 3，故 EPL = 3。这样，被引用的数据段描述子的 DPL 必须是 3，才能实现这次访问。若被引用的数据段描述子的 DPL = 2，尽管 CPL = 2，但由于 EPL = 3，这种访问也是不能实现的。

§15.5.4 描述子访问和特权检查

决定一个任务访问一个段的能力，要涉及到被访问的段的类型，所使用的指令、描述子的类型以及 CPL、RPL 和 DPL。段访问的两种基本类型是控制转移 (选择子装入 CS) 和数据访问(选择子装入 DS、ES 或 SS)。

一、数据段访问

把选择子装入 DS 和 ES 的指令，必须引用一个数据段描述子、或可读代码段描述子。任务的 CPL 和选择子的 RPL，必须是与描述子的 DPL，处于相同的特权层或更高的特权层，也就是说，一个任务只能访问特权层相同于、或低于 CPL、或 RPL 所指定的特权层的数据段，以防止一个程序访问它不能使用的数据。例如，当 CPL = 1 时，用 RPL = 2 的选择子，只能访问 DPL≥2 的描述子。

一般地，这个规则表示为在一个任务中 CPL、RPL 和 DPL 之间满足 max(CPL, RPL)≤

DPL时，程序对具有该DPL值的描述子的访问才能进行。

这个规则的一个例外是，可读协调代码段，这种类型的代码，可以被处于任何特权层中的任务读出。

如果特权检查失败(例如DPL数字上小于CPL和RPL中的最大者)，或引用了一个不正确的描述子类型(如门描述子或只执行代码段)，则将引起异常13。若段不存在，则将产生异常11。

把选择子装入SS的指令，必须引用可写数据段的数据段描述子。描述子特权(DPL)和RPL必须等于CPL。例如，在CPL=2时，装入SS寄存器的选择子的RPL和它所引用的描述子的DPL也必须等于2，即满足CPL=RPL=DPL，才能实现对作为堆栈段的描述子的访问。这是因为，在一个任务内，对每个特权层的操作，都提供了一个独立的堆栈的缘故。把其他类型的描述子，装入SS寄存器、或破坏上述的特权层规则，将引起异常13。一个段不存在错误，将引起异常12。

二、控制转移

当一个选择子被一个控制转移操作装入CS寄存器时，可能发生四种类型的控制转移。当然，每种转移只能发生在装入选择子的操作引用正确的描述子类型的情况下。它们的规则，见表15.15。任何违反这些描述子使用规则的操作，例如，通过一个调用门执行JMP或RET，把控制转移到任务状态段，将会引起异常13。

表15.15 关于控制转移的描述子访问规则

控制转移类型	操作类型	引用的描述子	描述子表
在同一特权层的段间转移	JMP, CALL, RET, IRET*	代码段	GDT/LDT
到相同或更高特权层的段间转移和任务内的中断，可以改变CPL	CALL	调用门	GDT/LDT
	中断指令，异常外部中断	自陷或中断门	IDT
到较低特权层的段间转移(改变任务的CPL)	RET IRET*	代码段	GDT/LDT
任务转换	CALL, JMP	任务状态段	GDT
	CALL JMP	任务门	GDT/LDT
	IRET** 中断指令，异常外部中断	任务门	IDT

* NT(标志字的嵌套任务位)=0

** NT(标志字的嵌套任务位)=1

控制转移可以分为任务内和任务间两大类。为控制转移而引用描述子的能力，与访问数据段时一样，也受到特权规则的支配。

(1) 任务内的控制转移

任务内的控制转移，是指在同一任务内，在同一特权层内的或在不同特权层之间的控制转移。

在同一特权层中的段间转移，如CALL或JMP指令，只能引用DPL等于任务CPL的代

码段描述子，或 DPL 的特权大于等于 CPL 的代码段描述子来引用一个协调代码段。当然，引用代码段描述子的选择子的 RPL，必须具有和 CPL 一样的特权。

在不同特权层之间的转移，必须引用 DPL≥CPL 的门描述子，也就是门描述子的特权要小于等于当前的任务特权。若 DPL 处于比 CPL 更高的特权层，则将引起异常 13。若该门中的目选择子，引用一个代码段描述子，则该代码段描述子的 DPL，必须具有相同于或大于任务 CPL 的特权，即要满足 DPL≤CPL。若不是这样，则产生异常 13。在控制转移以后，该代码段描述子的 DPL，就是任务的新的 CPL。若门中的目选择子，引用一个任务状态段，则系统将自动地执行一个任务转换。

与调用指令和进入中断处理相反，RET 和 IRET 指令，只能引用描述子特权小于等于任务的 CPL 的代码段描述子，也就是要满足 CPL≤DPL。在这种情况下装入 CS 的选择子，是从堆栈中返回的地址。在返回以后，选择子的 RPL，就是任务的新的 CPL。若 CPL 改变了，则老的堆栈指示器，在返回地址之后被弹出，这样，就恢复了原来特权层的堆栈。图 15 43 中的(a)和(b)，分别表示了在任务内同一特权层内和不同特权层间的控制转移情况。

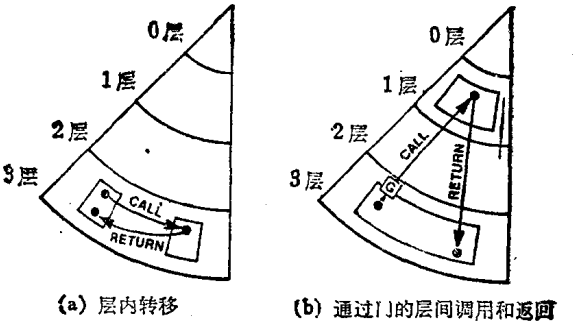


图 15.43 在一个任务内用 CALL 指令实现控制转移

(2) 任务间的控制转移

任务间的控制转移，必须引用一个处于和 CPL 相同、或较低特权层的任务门、或任务状态段，由任务门中的选择子，引用一个任务状态段，实现任务转换；或通过任务状态段的返回链，返回到原来调用该任务的状态段。图 15.44(a)、(b)，分别表示了从一个任务向另一个任务转移和从一个任务调用另一个任务的情况。

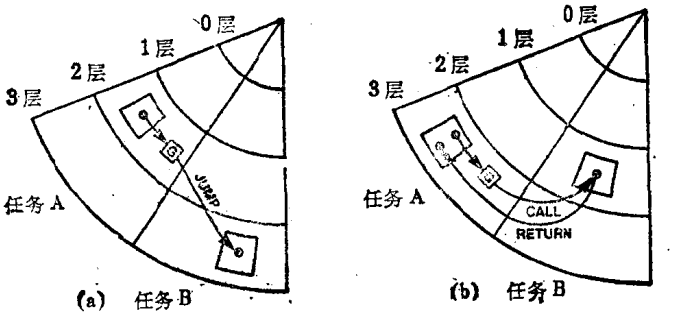


图 15.44 任务之间的控制转移

关于控制转移的特权规则，可以归纳为如下几条：

(1) 由 JMP 或 CALL 直接跳转或调用的代码段(代码段描述子)，只能是 DPL≤CPL 的

协调段，或是在同一特权层的 DPL = CPL 的非协调段；

(2) 在一个任务内的中断，或可以改变特权层的调用，只能通过一个在相同于或较低于 CPL 特权层的，即满足 CPL≤DPL 的门描述子，把控制转移到一个在相同于或大于 CPL 特权层的代码段；

(3) 不转换任务的返回指令，只能把控制返回到相同于或较低于 CPL 特权层的代码段；

(4) 任务转换可以通过调用、跳转或中断来实现，它们引用处于相同或较低特权层的任务门或任务状态段。

三、特权层的改变

任何改变任务的 CPL 的控制转移，也将引起堆栈的改变。处于特权层 0、1 和 2 的 SS:SP 的初始值，被保存在任务状态段中(参见任务转换操作部分)。在 CALL 控制转移期间，新的堆栈指示器，被装入到 SS 和 SP 中，而原来的堆栈指示器，则被推入到新的堆栈中去，以便在返回时，能正确地恢复在原来特权层的堆栈。当执行返回指令返回到原来的特权层时，原来的堆栈就被作为 RET(或 IRET))指令操作的一部分而被恢复。对于在堆栈上传递参数，并且穿越特权层次的子程序调用，在调用门中说明的参数的个数，被作为把参数从原来的堆栈，拷贝到新的堆栈的依据。对于这种调用，具有一个堆栈调整值的堆栈间 RET 指令，在返回时，将会正确地恢复以前的堆栈指示器。

§15.5.5 保　　护

一、对保护机构的要求

保护机构必须能保护系统的操作系统程序和数据，以防止应用程序对操作系统进行非法的或不恰当的修改，从而导致系统发生故障。系统中每个任务，也必须与其他任务的不合要求的作用相隔离。

系统模块之间控制的转移，必须是能精确控制的，以便取得高度的可靠性。系统必须能响应要求操作系统服务的请求，而不准越过操作系统把控制转移给调用程序。

保护机构必须能给系统中不同的程序指派不同的特权，对于较重要的程序给予较大的特权(如操作系统程序)。操作系统程序，总是被指派在较应用程序的特权层为高的特权层。

一个完整的保护机构，还必须能在寻址错误引起危险之前测得它们。为了防止错误编码的指令修改程序和执行数据，必须对每条指令进行测试，以证实它正在完成所期望的操作。

二、保护的实现

80286 含有对影响 CPU 执行状态的关键性指令 (例如 HLT)，进行测试和防止对代码和数据段的不恰当使用的保护机构。这些被称为"保护"的机构，具有三种形式：

(1) 限制段的使用 (如不许写只读数据段)。可供任务使用的，只能是在局部描述子表(LDT)和全局描述子表(GDT)中的描述子所定义的段；

(2) 通过使用描述子和特权规则，限止对段的访问；

(3) 特权指令或操作，只能在由 CPL 和 I/O 特权层 (IOPL) 决定的某些特权层中执行。IOPL 是由标志字的第 14 和 13 位定义的。

对所有的指令，保护机构都要执行上述的那些检查，它们可以分为段装入检查(表15.16)、操作数引用检查(表 15.17) 以及特权指令检查(表 15.18)。

表 15.16 段寄存器装入检查

错误说明	异常号数
超出描述子表的限	13
段描述子不存在	11或12
违反特权规则	13
无效的描述子/段类型的段寄存器装入： ——只读数据段装入SS ——特殊控制描述子装入DS, ES, SS ——只执行段装入DS, ES, SS ——数据段装入CS ——读/执行代码段装入SS	13

表 15.17 操作数引用检查

错误说明	异常号数
写入代码段	13
从只执行代码段中读	13
写入只读数据段	13
超出段限*	13或12

* 偏移量计算中的进位忽略

表 15.18 特权指令检查

错误说明	异常号数
当CPL≠0时执行下列指令： LIDT, LLDT, LGDT, LTR, LMSW, CTS, HLT	13
当CPL>IOPL时执行下列指令： INS, IN, OUTS, OUT, STI, CLI, LOCK	13

任何违反上面三个表所表示的规则的操作，将会引起一个异常。一个与堆栈段不存在相关的异常，将会引起产生异常 12。若 CPL 不具有足够的特权（数字上足够小），则 IRET 和 POPF 指令，将不执行它们的某些功能。当这种情况发生时，不产生异常或其他说明。

若 CPL > IOPL，则 IF 位不改变。

若 CPL > 0，则标志字的 IOPL 字段不改变。

三、异常

在这一章中，我们已经在许多地方接触到“异常”这个词。正如我们已经知道的，异常是 80286 支持的一种内部中断，这种内部中断的特点是：在中断处理之后，它返回的地址是引起异常的指令(包括任意的可能有的指令前缀)的地址，从而提供了在异常条件消失(经过异常中

断处理使异常条件消失)之后,再次执行该条指令的能力,这是对执行中的程序,实行保护的有力工具。

80286 在保护方式下,能检测到若干种类的异常和中断,其中大多数在异常条件消失以后,是可重新启动的。表 15.19 说明了这种情况。

表 15.19 保护方式异常

中断向量	功能	返回到失败指令的地址吗?	总是可重新启动的吗?	错误码在堆栈中吗?
8	检测到双重异常	返回	否	是
9	协处理器段越限	不返回	否	否
10	无效任务状态段	返回	是	是
11	段不存在	返回	是	是
12	堆栈段越出或段不存在	返回	是*	是
13	一般保护	返回	否	是

* 当 PUSHA 或 POPA 指令试图首尾连接堆栈段时,执行后的机器状态,将是不可重新启动的。这种情况是由被保护的 SP 是 0000H、0001H、0FFFEH 或 0FFFFH 来识别的。

§15.5.6 特殊操作

一、中断

实时交互计算机系统,必须处理频繁的中断。80286 提供了一个高速的中断响应机构,使之在处理大量中断的同时,仍有最大的吞吐量。80286 能在少于 4μs 的时间内,响应一个中断,从而支持了中断的迅速处理。在系统内部,这样少的等待时间,使系统能支持更实时的事件。

保护方式的中断系统,包含两种转换到中断服务子程序的方法。第一种支持驻留在被中断任务的地址空间中的中断服务程序。在全局地址空间中,这些程序通常是操作系统的一部分。第二种中断转移,自动地完成从被中断的任务到指定的中断服务任务的高速任务转换,所指定的中断服务任务,是与其他任务完全隔离的。第二种中断,为在现有的系统中增加新的特性,或定制的 I/O 驱动器提供了一种方便的手段。

这些中断操作,对软件来说是透明的。两种中断转移,使用相同的从中断返回的指令格式,以便将控制返回到被中断的任务。

二、任务转换操作

在多用户或多任务系统中,操作系统最重要的功能之一,就是任务调度。为了允许多道程序的并行运行,操作系统必须能支持多任务,而要支持多任务,就要进行频繁的任务转换操作。

80286 具有片内的高速硬件,来完成这种操作。10 MHz 的 80286,能在 17μs 内保护一个任务的状态(所有寄存器的内容),装入另一个任务的状态(为所有的寄存器送入对应于该任务的值),并恢复运行。为了取得较高的性能,80286 还允许中断直接引起任务转换,而不需操作系统干预。

任务转换操作，是由执行段间的JMP、或CALL指令来调用的。这些指令引用一个任务状态段TSS(Task State Segment)，或在GDT或LDT中的任务门描述子。INT$_n$指令、异常或外部中断，通过选择一个相应的IDT项目中的任务门，也能调用任务转换操作。

TSS描述子，指着一个包含着全部80286的执行状态的段(见图15.45)，而一个任务门描述子，则包含着一个TSS选择子。其中的限字段必须≥002BH。

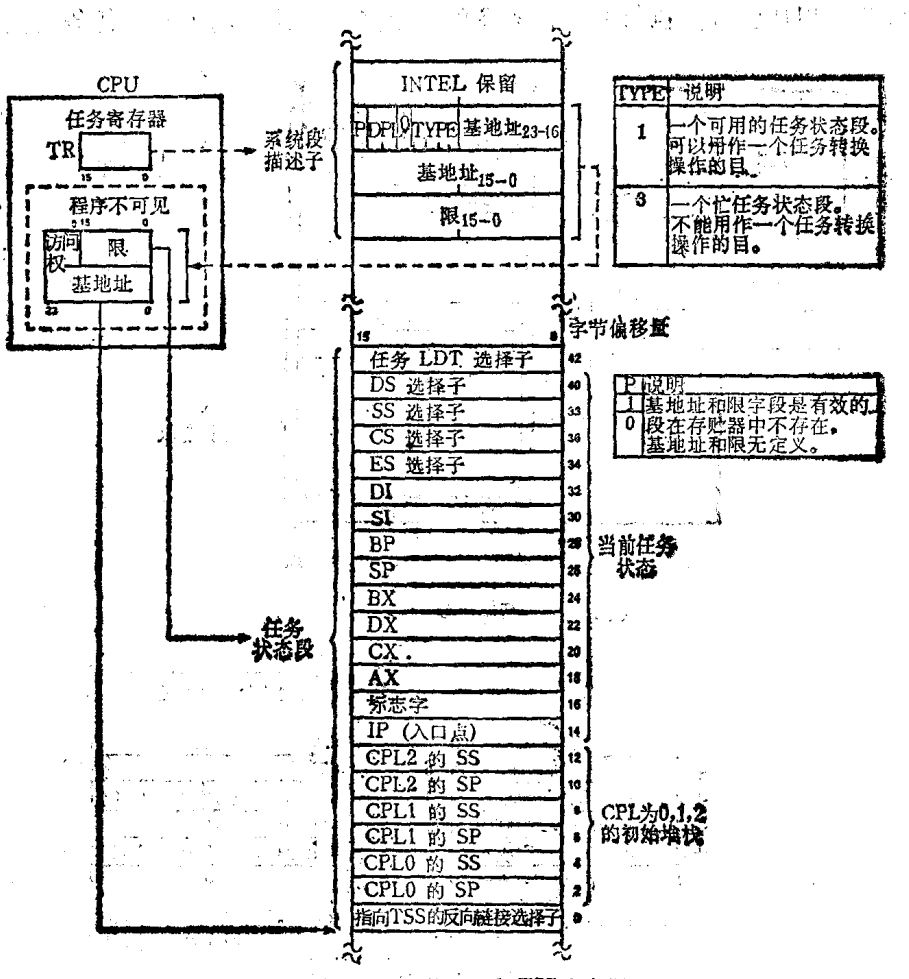


图 15.45 任务状态段和TSS寄存器

每个任务都有一个TSS。当前的TSS，由一个在80286内部的专用寄存器TR(Task Register)来识别。这个寄存器含有一个引用当前TSS的任务状态段描述子的选择子。与TR寄存器相联系着的还有一个隐含的24位基地址寄存器、和一个隐含的16位的限寄存器。每当TR被装入一个新选择子时，这些寄存器也被自动地装入与TR中选择子相对应的任务状态段的基地址和限。这些寄存器，就是80286对任务转换操作所提供的硬件支持。

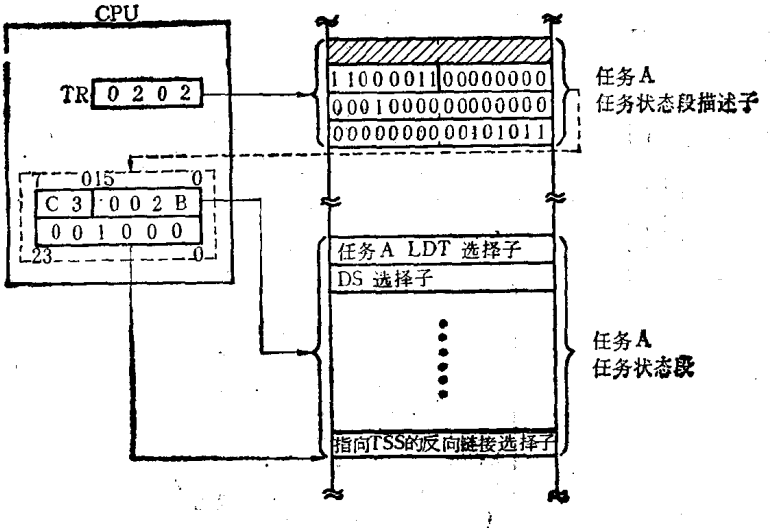
IRET指令，被用来把控制返回到调用当前任务的任务、或被中断的任务。在标志寄存器中的第14位称为嵌套任务位NT(Nested Task)，它控制着IRET指令的功能。若NT=0，则IRET指令执行正常的当前任务的返回操作。若NT=1，则IRET执行一个任务转换操作，以返回到调用该任务的任务。

当一条CALL或INT指令，启动一个任务转换操作时，老的和新的TSS，将被标记为忙(通过把任务状态段描述子的TYPE，从1改为3来标记忙)，并且新TSS的向后链接字段，被设

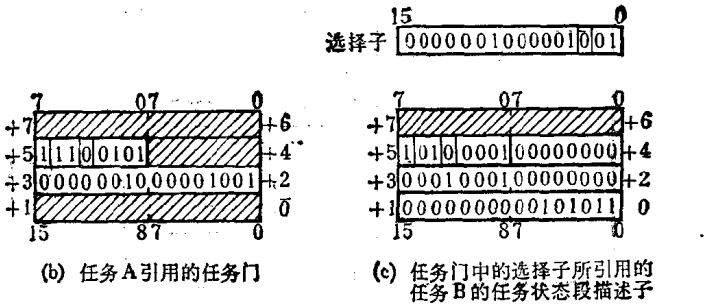
置为指向老的TSS选择子。新任务的NT位被CALL或INT启动的任务转换位置。不引起任务转换的中断，将清除NT位。NT位也可以被POPF或IRET指令设置或清除。

使用一个引用处于忙状态的任务状态段的选择子，会导致异常13。

我们用一个例子，来说明任务转换操作的大致过程。假定任务A的CPL=2，在某一时刻，任务状态段和TSS寄存器的状态如图15.46(a)所示。现在任务A，要通过一个如图15.46(b)所示的任务门调用任务B。执行调用任务B的指令之后，就启动了一个任务转换操作。处理



(a) 任务A的初始状态



(b) 任务A引用的任务门

(c) 任务门中的选择子所引用的任务B的任务状态段描述子

图15.46 任务转换操作(I)

器先用图(b)所示的任务门中给出的信息，在GDT表中找到变址值为41的项，在这一项中放着如图15.46(c)所示的任务B的任务状态段描述子。任务转换机构，把任务A的各寄存器的当前值，保护入任务A的任务状态段，然后，把任务B的任务状态段选择子，送入TR寄存器，并把该选择子所指的描述子的TYPE置为3。由于TR寄存器的被装入，在描述子中的基地址和限，以及访问权字节的信息，也被自动地装入到相应的寄存器中。在完成了任务B的任务状态段向任务A的任务状态段描述子的连接，把NT标志置位为1，并把任务B的任务状态段中的内容，恢复到各寄存器以后，就可以开始任务B的执行了。这时任务转换机构和存贮器中的内容，如图15.47所示。

注意，在这个例子中，任务A的CPL=2，它引用的任务门的DPL=3，满足CPL≤DPL的条件，故访问是允许的。任务B的任务状态段描述子的DPL=1，表示它要在CPL≤1时才能被访问。当从任务B的任务状态段中，恢复CS寄存器值的时候，就建立了新的当前任

务特权层。这样，既实现了任务的转换，同时也通过门描述子，实现了特权层的改变。当任务B完成之后，就会执行一条IRET指令来返回，由于NT标志为1，所以这个操作会通过任务B的任务状态段中的反链信息，把控制返回到任务A。

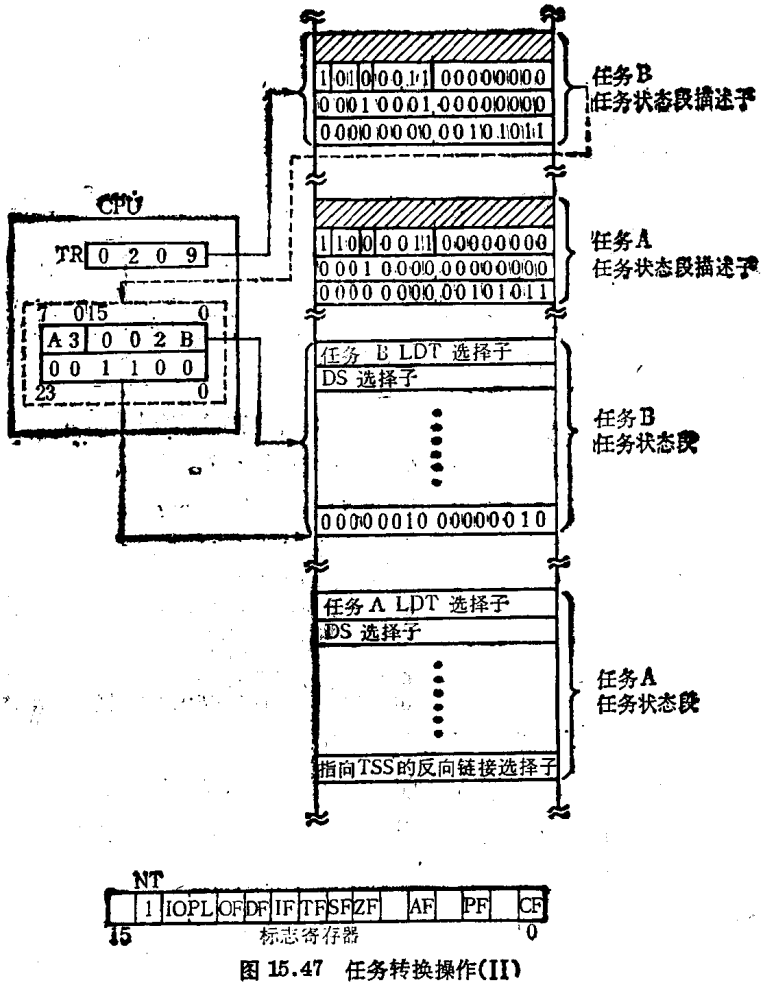


图 15.47 任务转换操作(II)

三、虚拟存贮器

任何支持虚地址空间的计算机系统，必须有效地管理系统内存和外存(通常为盘)之间的程序(和数据)的交换。假如一条指令引用一个在内存中不存在的虚地址，系统必须具有一种方法去通知操作系统，将所要求的实体调入内存。

以前关于描述子装入的讨论，总是假定目标段是在内存中的。当操作系统用完了内存空间以后，为了把需要的段从外存调入内存，它就先要从内存中把一个或多个段调到外存去，以产生可供新的段调入的内存空间。为了使操作系统能容易地识别出最佳的交换候选段，80286在每次对一个段访问时，都把该段标记为"访问过"。这是通过把段描述子访问权字节中的A字段置位来实现的。

在操作系统确定了那个段使用最少之后，它便把该段移到外存中去，并把该段描述子访问权字节中的P字段复位，以表示该段在内存中不存在。没有被访问过的段，可以简单地进行重写，因为在外存中的副本和当前内存中的段是一样的。

若一条指令请求访问一个在内存中不存在的段，该段描述子中的不存在位，就会触发引起一个段不存在异常。操作系统是这样处理这类中断的，它腾出所要求数量的实地址空间，然后把目标段从盘中调入内存(见图 15.48)。在段装入到内存中以后，操作系统把该描述子的P字段置位为1，以表示该段存在于内存中，同时对该描述子中的地址信息进行更新。这些操作完成之后，也就是异常处理完成之后，操作系统就可以重新启动最初请求访问该段的指令。

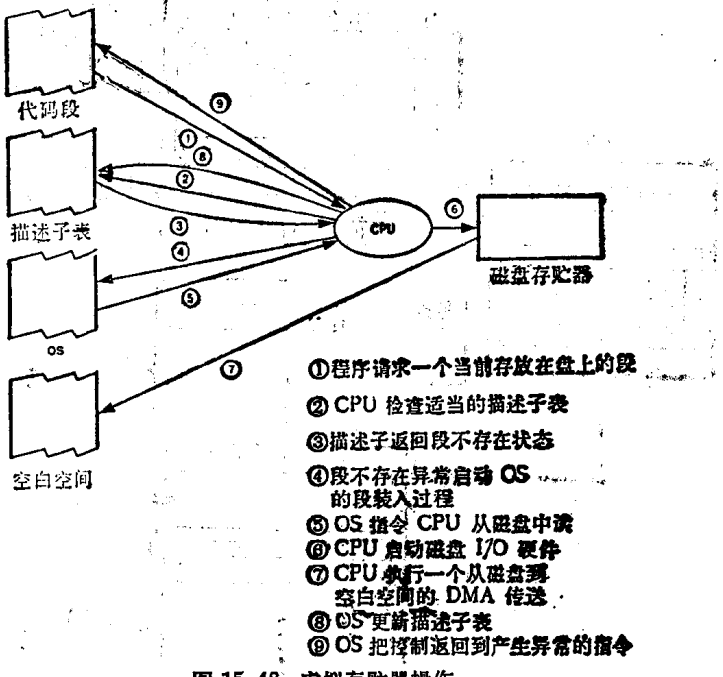


图 15.48 虚拟存贮器操作

由于 80286 能在执行引起异常的指令之前，检测到异常，因此，指令的重新启动是非常简单的。处理器只要把被中断指令的地址放回到指令指示器中，并继续执行程序就可以了。

四、可恢复的堆栈故障

在支持多道程序的系统中的常见问题是堆栈溢出。这种情况起因于大量应用递归过程或子程序，这些递归过程或子程序，往往要求堆栈提供更多的堆栈上的动态存贮单元。大多数系统中堆栈溢出的结果不是破坏信息，就是导致系统故障。80286 的保护机构，能防止这些灾难的发生。

象上面所讨论的那样，80286 实际上是在引起堆栈溢出的指令的执行之前，就暂停运行该指令。因为指令尚未执行，所以在异常处理程序纠正这个问题之后，重新启动就很简单。

对于试图超越现行堆栈段的限的写入指令 80286 能在问题产生之前，就检测到这种异常情况，在由此引起的异常处理中，就可以调用增加堆栈段长度的过程。在异常处理完之后，处理器重新启动被中断的指令，于是，原来的程序，就能在一个经过扩充的堆栈的环境下运行。80286 也能检测潜在的堆栈下溢出。

五、协处理器上下文转换

协处理器(如 80287 数字处理器)的上下文关系，并不为任务转换操作所改变。协处理器的上下文关系，只在一个不同的任务试图要使用该协处理器(它仍然包含着前一个任务的上下

文关系)时,才需要改变。80286 在任务转换之后,用引起协处理器不存在异常的办法,来检测协处理器的第一次使用。

每当 80286 转换任务时,它总是设置 MSW 的 TS(Task Switched)位。TS 表示一个协处理器的上下文可以属于另一个任务,而不是当前的任务。若 TS = 1,并且协处理器是存在的(MSW 中的 MP = 1),此时若试图执行 ESC 或 WAIT 指令,就会引起协处理器不存在异常(7)。

六、指示器测试指令

80286 提供若干指令来加速为保持系统的完整性而进行的指示器测试和连贯性检查。这些指令使用存贮器管理硬件,来证实一个引用适当的段的选择子值不会有引起异常的危险。一个条件标志指示着使用该选择子或段是否会引起一个异常。表 15.20 是指示器测试指令。

表 15.20 指示器测试指令

指　令	操 作 数	功　能
ARPL	选择子寄存器	Adjust Requested Privilege Level(调整所请求的特权层):把选择子的 RPL 调整到当前选择子 RPL 值和在寄存器中的 RPL 值的最大数字。若选择子的 RPL 被改变了,就设置 0 标志 ZF。
VERR	选 择 子	VERify for Read (读检验):若选择子引用的段可以被读,就设置 0 标志 ZF。
VERW	选 择 子	VERify for Write(写检验):若选择子引用的段可以被写,就设置 0 标志 ZF。
LSL	寄存器选择子	Load Segment Limit(装入段限):若特权规则和描述子类型允许,就把段的限读入寄存器。若成功,就设置 0 标志 ZF。
LAR	寄存器选择子	Load Access Right(装入访问权):若特权规则允许,就把描述子访问权字节读入寄存器。若成功,就设置 0 标志 ZF。

七、双重错误和停机

若在一条指令的执行中,检测到两个分离的异常,80286 就执行双重错误异常(8)。若在双重错误异常的处理过程中,又发生了一个异常,则 80286 将进入停机。在停机期间,80286 不再处理下面的指令或异常。NMI(CPU 留在保护方式中)或 RESET(CPU 终止保护方式)都能迫使 80286 退出停机状态。停机在外部是 HALT 总线操作用 A_1 为高电平通知的。

八、保护方式初始化

在 RESET 之后,80286 最初是在实地址方式下运行的。为了能把初始化代码,放在物理存贮器的顶部,80286 一开始用 CS 寄存器进行存贮器引用时,A_{23}—A_{20} 将是高电平,这个过程一直延续到 CS 被改变为止,引用 DS、ES 或 SS 段时,A_{23}—A_{20} 将是低电平。在实地址方式中,启动以后 CS 寄存器的任何变化,都会迫使 A_{23}—A_{20} 变成低电平,并且在以后用 CS 寄存器引用存贮器时,始终为低电平。初始的 CS:IP 值 FF00:FFF0,为初始化代码,提供了不需要改变 CS 的 64K 字节的代码空间。

在把 80286 放入保护方式之前,几个寄存器必须被初始化。GDT 和 IDT 基地址寄存器必须引用有效的 GDT 和 IDT。在执行 LMSW 指令设置 PE 位以后,80286 必须立即执行一条

段内 JMP 指令，来清除在实地址方式下，已被译码的指令队列。

为要迫使 80286 寄存器与由软件假定的初始保护方式的状态相匹配，就要执行一条具有引用在系统中使用的初始 TSS 的选择子的 JMP 指令。这样，就将装入任务寄存器、局部描述子表寄存器、段寄存器和初始的通用寄存器状态。TR 寄存器应当指向一个有效的 TSS，因为一个任务转换操作，要保护当前的任务状态。

§15.5.7 保护虚地址方式的总结

从以上的介绍中可以看到，iAPX286 的保护虚地址方式，既继承了实地址方式的特点，又

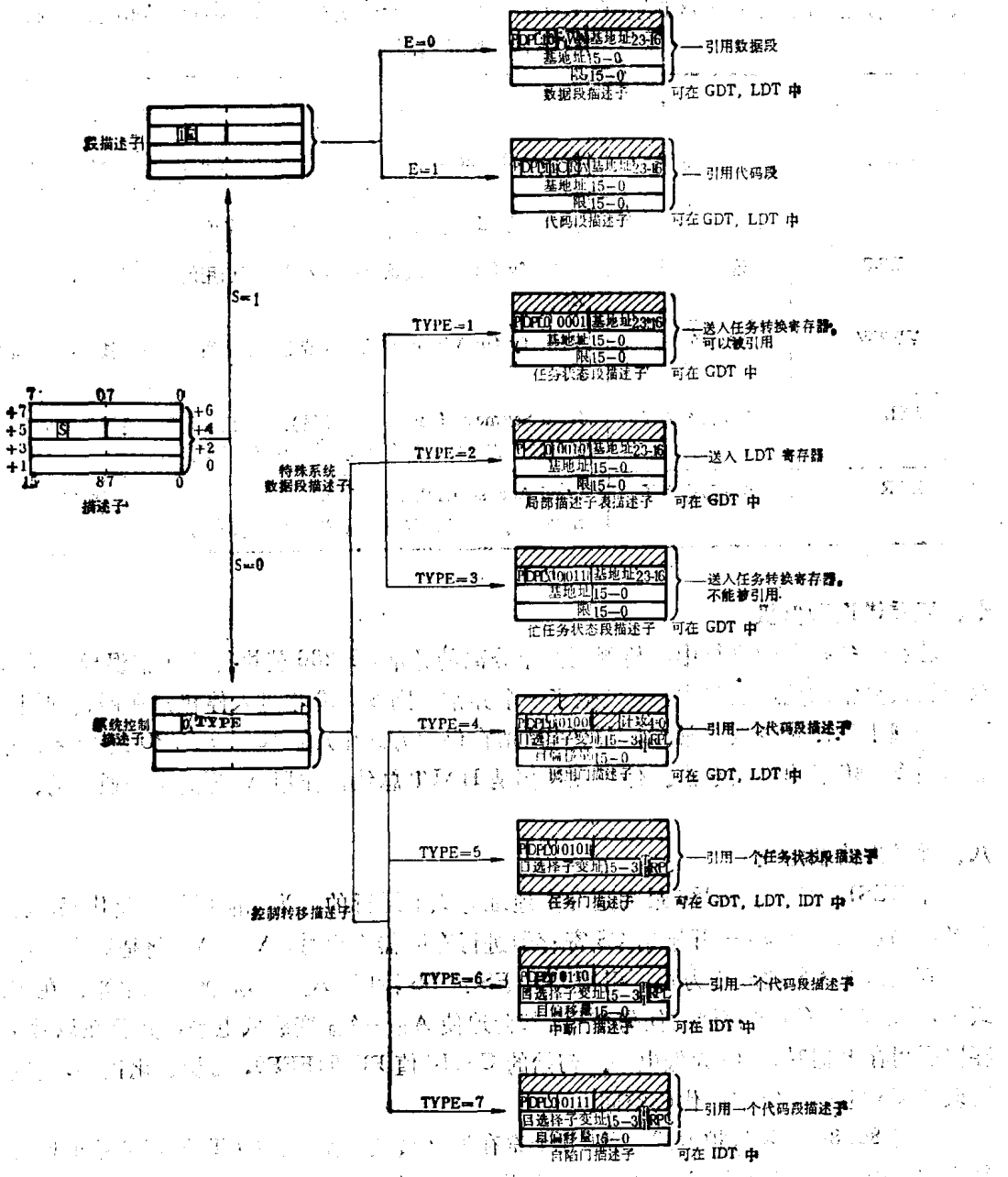


图 15.49 iAPX286 描述子总结

与实地址方式有很大的不同。保护虚地址方式，主要是以所谓描述子和选择子的数据结构、并配以相应的硬件支持来实现的。各种描述子具有差不多相同的格式，集中了系统控制和管理信息。选择子改变了在实地址方式中作为一个基地址的高16位的形式，而变成了进入一个描述子表的变址，并且含有引用那一个描述子表和所要请求的特权层的信息。由于提供了描述子、选择子这一缓冲层，一方面可以方便地构成24位物理地址，另一方面也可以在对这些数据结构进行操作的时候，方便地进行特权检查，为操作系统和任务分离提供了物质基础，实现任务之间和程序之间的保密，也有了可能。

由80286硬件实施的特权机构，提供了在代码和数据段使用期间的全部控制。程序只能访问处于相同的或更低特权层上的数据，并且只能调用处于相同或更高特权层上的服务程序。

关于特权规则，可以简单地归纳如下：

(1) 当前任务特权层CPL，是由CS寄存器的最低两位表示的，实际上是由装入CS的选择子的RPL决定的；

(2) 描述子特权指定了一个可以访问该描述子的最不可靠的CPL特权层，即要CPL≤DPL，才能访问；

(3) 在使用选择子时，它的RPL和当前的CPL中的数值较大者，构成有效特权层EPL。EPL能在CPL的基础上，进一步缩小特权；

(4) 当一个选择子被控制转移操作装入CS时，发生四种类型的控制转移，可以改变CPL，但也受到特权规则的限制。在控制转移以后，代码段描述子的DPL，就是任务的新的CPL。

为了使读者对描述子这一数据结构有一个比较全面的了解，我们在图15.49中表示了描述子之间的关系和它们的结构。

§15.6 系统接口

80286的系统接口，以局部总线和系统总线二种形式出现。局部总线由80286引脚上的地址、数据、状态和控制信号组成。系统总线是任意局部总线的缓冲形式。系统总线也可依据状态和控制线的编码和/或信号的定时和信号的装入而不同于局部总线。iAPX286系列，包含有若干种产生标准系统总线的设备，象IEEE 796标准的Multibus™等。

§15.6.1 总线接口信号和定时

iAPX286微系统局部总线，把80286与局部存贮器和I/O部件相连接。接口有24根地址线，16根数据线和8种状态与控制信号。

80286 CPU、82284时钟发生器、82288总线控制器、82289总线仲裁器、8286/7收发器和8282/3锁存器，提供了一个缓冲的和译码的系统总线接口。82284产生系统时钟，同步$\overline{\text{READY}}$和RESET信号。82289总线仲裁器产生多路的总线仲裁信号。这些部件为包含多总线在内的大多数系统总线接口，提供所需要的定时和电功率驱动层。

§15.6.2 物理存贮器和I/O接口

在保护方式中，可寻址物理存贮器的最大容量为16兆字节，在实地址方式中可寻址1兆字节。存贮器可以按字节或字访问。一个字可由任意两个连续可寻址的字节组成，并且低位字节存放在低地址中。

字节传送，发生在16位数据总线的高8位部分或低8位部分。偶地址字节通过D_{7-0}访问，奇地址字节则通过D_{15-8}访问。偶地址字通过D_{15-0}在一个总线周期内进行传送，奇地址字则需要两个总线周期，第一个周期在D_{15-8}上传送数据，第二个周期在D_{7-0}上传送数据。这两个字节的传送是自动进行的，对于软件来说是透明的。

A_0和$\overline{BHE}$这两个总线信号，用来控制在数据总线的高8位还是低8位部分上传送数据。偶地址字节传送是由A_0为低电平和$\overline{BHE}$为高电平确定的，奇地址字节传送，则是由A_0为高电平和$\overline{BHE}$为低电平来确定的。A_0和$\overline{BHE}$均为低电平，表示偶地址字传送。

在两种方式中，I/O地址空间均含有64K字节地址。和在存贮器中的情况一样，I/O空间可按字或字节访问。8位的外围设备，既可接到数据总线的高8位字节，也可接到低8位字节。接在高8位数据总线(D_{15-8})的8位I/O设备，是按奇I/O地址访问的。接在低8位数据总线(D_{7-0})的设备，是按偶地址访问的。8259A中断控制器，必须连接在低8位数据总线(D_{7-0})上，以便返回正确的中断向量。

§15.6.3 总线操作

80286使用二倍频系统时钟(CLK输入)，来控制总线定时。局部总线上所有的信号，都是相对于系统CLK来计量的。CPU把系统时钟除以2，来产生处理器时钟，并由它来确定总线状态。每一处理器时钟，由两个系统时钟周期组成，即节拍1和节拍2。82284时钟发生器的输出(PCLK)，确定处理器时钟的下一节拍。系统时钟和处理器时钟之间的关系，在图15.50中表示。

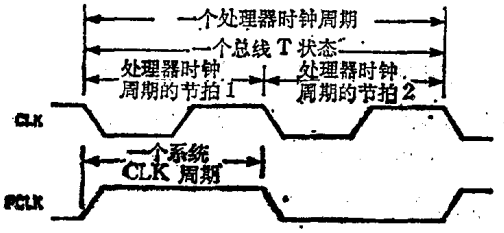


图15.50 系统时钟和处理器时钟之间的关系

80286支持六种类型的总线操作：存贮器读、存贮器写、I/O读、I/O写、中断响应和暂停/停机。数据传输的最大速率，为每两个处理器时钟周期传送一个字。

iAPX 286总线有三种基本状态：空闲(T_i)、发送状态(T_s)和执行命令(T_c)。80286 CPU还有第4种局部总线状态，称为保持(T_h)，T_h表示80286在响应HOLD请求，已把局部总线的控制权转让给其他的总线主设备。

每个总线状态，为一个处理器时钟长。图15.51表示了4种局部总线状态和允许的转换。

§15.6.4 总线状态

T_i空闲状态，表示没有数据传送正在被处理或请求。第一个有效状态T_s，是由状态线$\overline{S_1}$或$\overline{S_0}$变低电平通知的，同时也表示处理器时钟的节拍1。在T_s期间，指令编码、地址和数

据（对写操作来说），在80286输出引脚上是有效的。82288总线控制器，对状态信号进行译码，并产生多总线兼容的读/写命令和局部收发器控制信号。

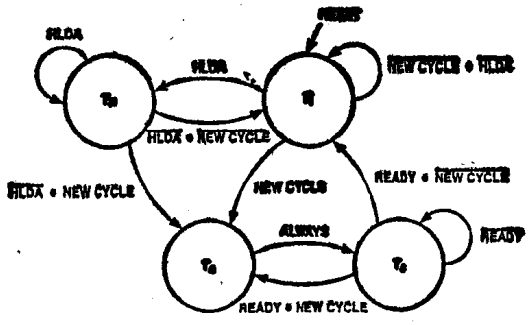


图 15.51 80286总线状态

T_s之后，进入执行命令状态(T_c)。在T_c期间，存贮器和I/O设备响应总线操作，完成传送数据到CPU或接受写入的数据。为了确保存贮器或I/O设备有足够的时间作出响应，T_c状态常常是可以重复的。是否需要重复T_c状态，由$\overline{\text{READY}}$信号决定。

在T_h保持状态期间，80286将浮空所有的地址、数据和状态输出引脚，以便使另外的总线主设备能使用局部总线。80286的HOLD输入信号，用来使80286进入T_h状态。80286的HLDA输出信号，表示CPU已经进入了T_h状态。

§15.6.5 流水线寻址

80286使用具有流水线定时的局部总线接口，以便使数据访问，有尽可能多的时间。流水线定时，使总线操作能在2个处理器周期内完成，而每个单独的总线操作，都要延续3个处理器周期。

地址输出的定时，也是流水线的，这样可使下一总线操作的地址，在当前总线操作期间就变为可供使用。换句话说，下一总线操作的第一个时钟周期与现行总线操作的最后一个时钟周期是重叠的。因此，下一总线操作的地址译码和路径选择逻辑，可以在下一总线操作之前进行操作。外部的地址锁存器，为整个总线操作保持稳定的地址，并提供附加的AC和DC缓冲。

在所有的T_c状态期间，80286不保留现行总线操作的地址。相反，在任何一个T_c的节拍2期间，下一总线操作的地址便可发出。在第1个T_c的节拍1期间，地址保持有效，以便确保与ALE相关的地址锁存器输入的保持时间。图15.52，是80286的基本总线周期。

§15.6.6 总线控制信号

82288总线控制器提供控制信号，如地址锁存允许(ALE)、读/写命令、数据发送/接收(DT/$\overline{\text{R}}$)和数据允许(DEN)等信号，它们为存贮器和I/O系统控制着地址锁存、数据收发、写允许和输出允许等操作。

地址锁存允许(ALE)输出确定地址在什么时候可以被锁存。从前一总线操作结束，到下一总线操作地址在锁存器输出端出现，ALE提供了至少一个系统时钟周期的地址保持时间。这个地址保持时间，是支持多总线和公共存贮器系统所需要的。

数据总线收发器，是由 82288 输出的数据允许(DEN)和数据发送/接收(DT/$\overline{R}$)信号来控制的。DEN 启动数据收发器，而 DT/$\overline{R}$ 则控制收发器的方向。DEN 和 DT/$\overline{R}$ 被定时得能防止在总线主设备、数据总线收发器和系统数据总线收发器之间争用总线。

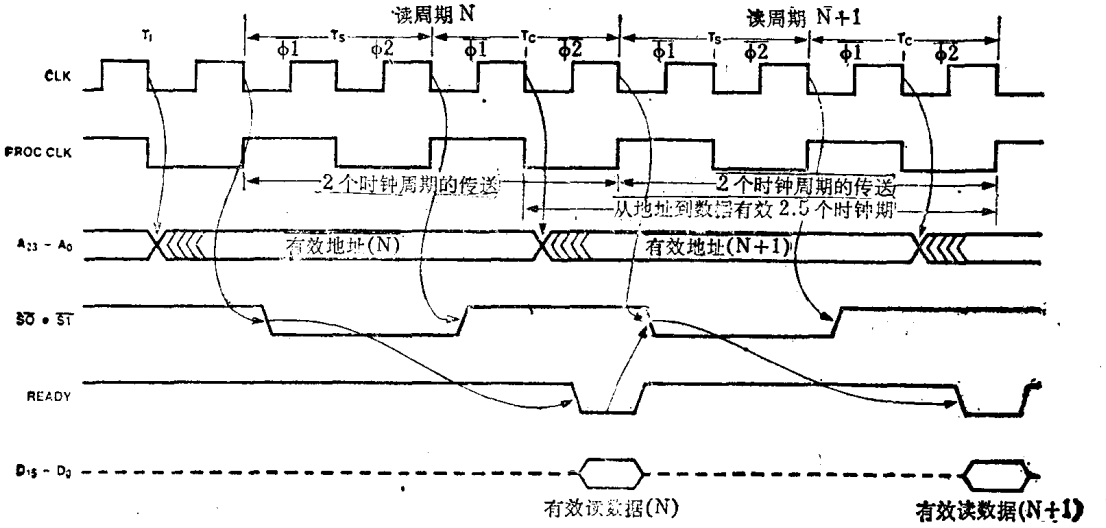


图 15.52　80283 基本总线周期

§15.6.7　命令定时控制

iAPX286 局部总线，提供两种系统定时预选，即命令扩展和命令延迟。

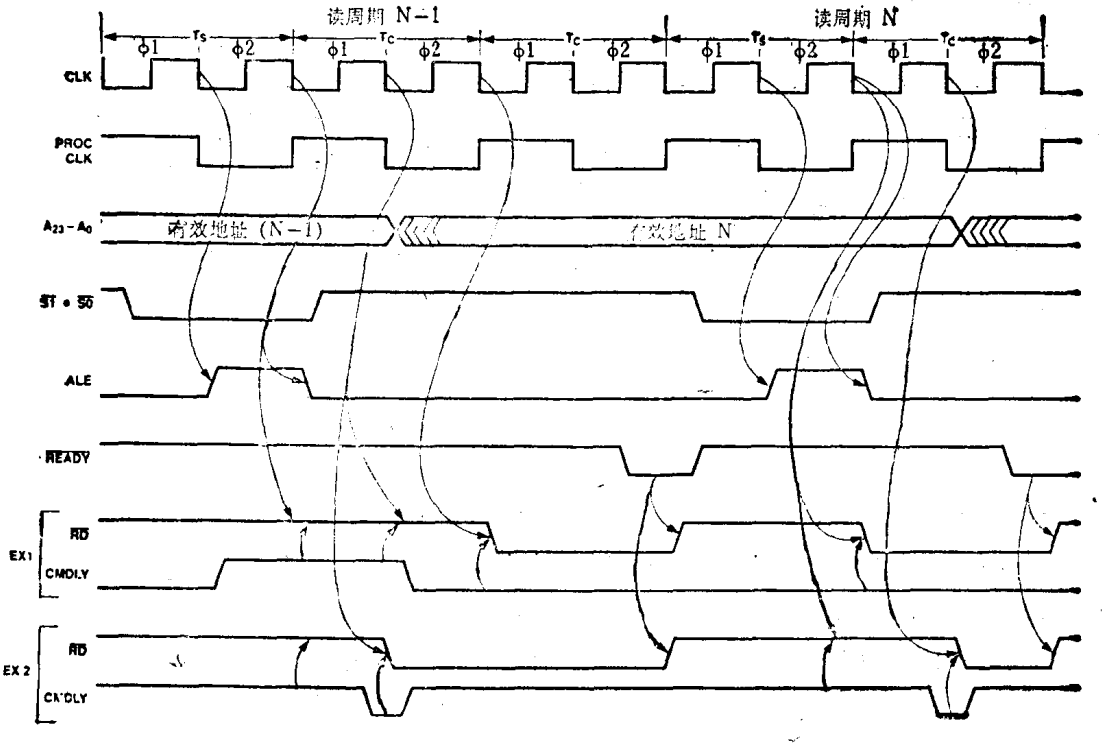


图 15.53　CMDLY 控制和命令的前沿

令命扩展使外部设备有足够的时间去响应命令，类似于8086的插入等待状态。外部的逻辑能控制任意总线操作的持续时间，这样可使总线操作只占用必需的时间。$\overline{\text{READY}}$输入信号能按需要延长任意总线操作。

命令延迟在系统总线命令要变为有效时，利用延时手段，加上一个地址或写数据建立时间，以延迟任何总线操作的系统总线命令生效。命令延迟是由82288的CMDLY输入来控制的。在T_s之后，总线控制器在CLK的每一个下降沿对CMDLY采样，若CMDLY为高电平，则82288将不激活命令信号，也就是进行延迟；而当CMDLY为低电平时，82288就将激活命令信号。在命令变成有效以后，CMDLY就不再被采样。

当一个命令被延迟时，从命令生效，到返回读数据、或接收写数据之间的响应时间就减少了。为了预定系统总线定时，可以用一个地址译码器，来决定那一个总线操作需要将命令进行延迟。CMDLY输入不影响ALE、DEN或$DT/\overline{R}$的定时。

图15.53，表示了CMDLY的4种应用。E_x1表示对周期N-1延迟读命令2个系统时钟周期，对周期N不延迟。E_x2表示对周期N-1延迟读命令一个时钟周期，对周期N也延迟一个时钟周期。

§15.6.8 总线周期结束

在最大的传输速率下，iAPX286总线在T_s和T_c状态之间交替。总线状态信号在T_s之后变成无效，以便在当前周期完成以后，正确地发出下一总线操作开始的信号。在iAPX286局部总线上不存在T_c的外部指示。总线主设备和总线控制器在T_s之后，直接进入T_c，并继续运行T_c周期，直到由$\overline{\text{READY}}$结束为止。

§15.6.9 $\overline{\text{READY}}$ 操 作

当前总线主设备和82288总线控制器，同时结束每一个总线操作，以达到最大的总线带宽。它们均由$\overline{\text{READY}}$变有效而预先通知，$\overline{\text{READY}}$标识现行总线操作的最后一个T_c周期，总线主设备和总线控制器必须收到同一个$\overline{\text{READY}}$信号，因此，要求$\overline{\text{READY}}$与系统时钟同步。

一、同步准备就绪

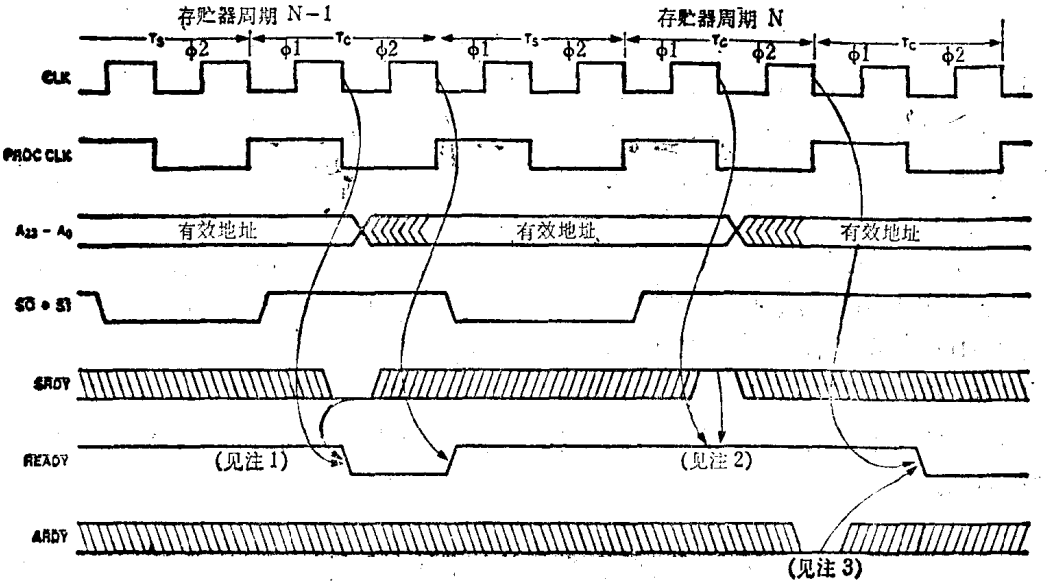
82284时钟发生器，接收来自同步和异步源的信号，而提供$\overline{\text{READY}}$同步信号（参见图15.54）。时钟发生器的同步准备就绪输入($\overline{\text{SRDY}}$)，在每个T_c的节拍1的下降沿被采样，然后，SRDY的状态通过READY输出线，传送到总线主设备和总线控制器。

二、异步准备就绪

许多系统，带有与系统时钟异步的设备或子系统。因此，它们的准备就绪输出，不能保证满足82284 $\overline{\text{SRDY}}$的建立和保持时间的要求。82284异步准备就绪输入($\overline{\text{ARDY}}$)，就是专门用来接收这种信号的。82284的同步逻辑，在每个T_c周期的开始时采样$\overline{\text{ARDY}}$输入。这样，在把它传送到总线主设备和总线控制器之前，有一个系统CLK周期的时间，来判定它的值。

82284对每种准备就绪输入，有一个启动引脚($\overline{\text{SRDYEN}}$和$\overline{\text{ARDYEN}}$)，它们用来选择当前总线操作是否将由同步或异步准备就绪信号来结束。任何一个准备就绪输入，均可以结束一

个总线操作.这些允许输入信号,是低电平有效的,并且与它们各自的准备就绪输入,有着相同的定时. 地址译码逻辑，通常作出当前总线操作应该由$\overline{\text{ARDY}}$还是$\overline{\text{SRDY}}$来结束的选择.



注:
1. $\overline{\text{SRDYEN}}$是低电平有效
2. 若$\overline{\text{SRDYEN}}$为高电平,则$\overline{\text{SRDY}}$的状态将不影响$\overline{\text{READY}}$
3. $\overline{\text{ARDYEN}}$是低电平有效

图 15.54 同步和异步的准备就绪

§15.6.10 数据总线控制

图 15.55——图 15.57,显示了 DT/$\overline{\text{R}}$、DEN、数据总线和地址信号是如何对读、写、空闲等

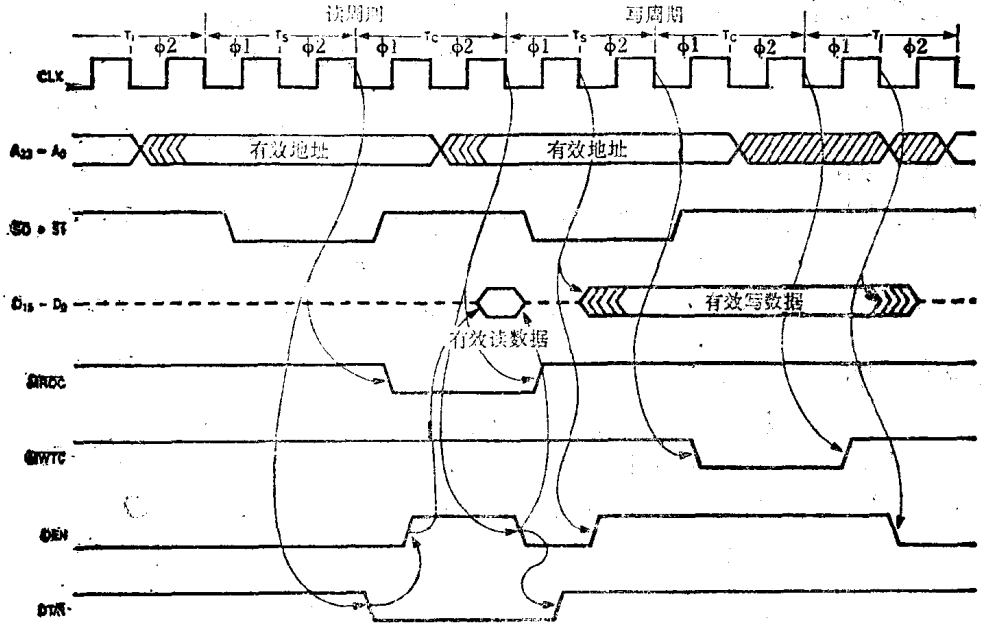


图 15.55 背对背的读—写周期

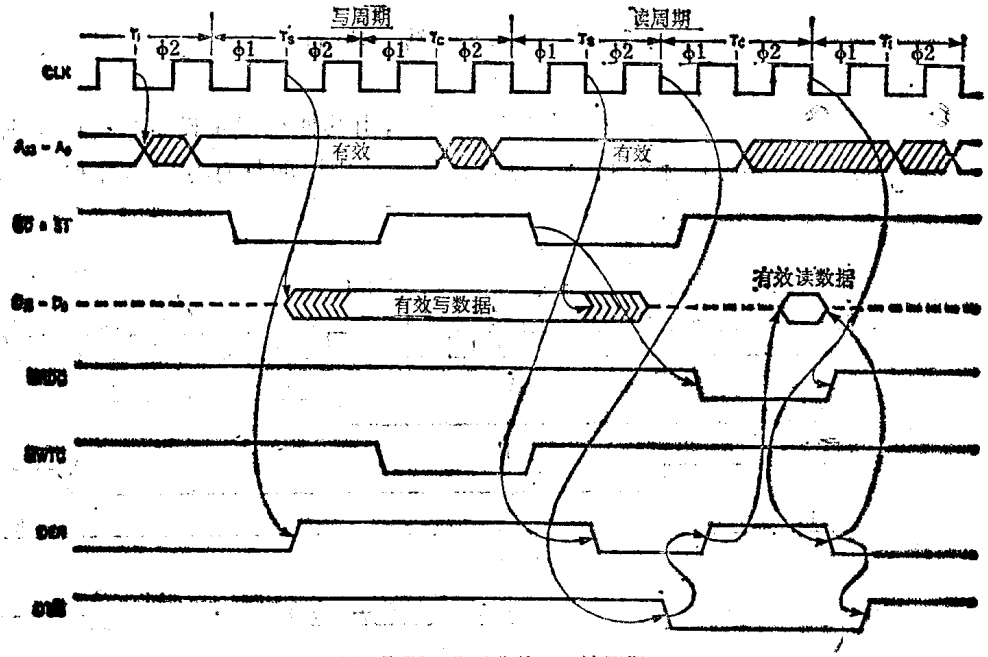


图 15.56 背对背的写—读周期

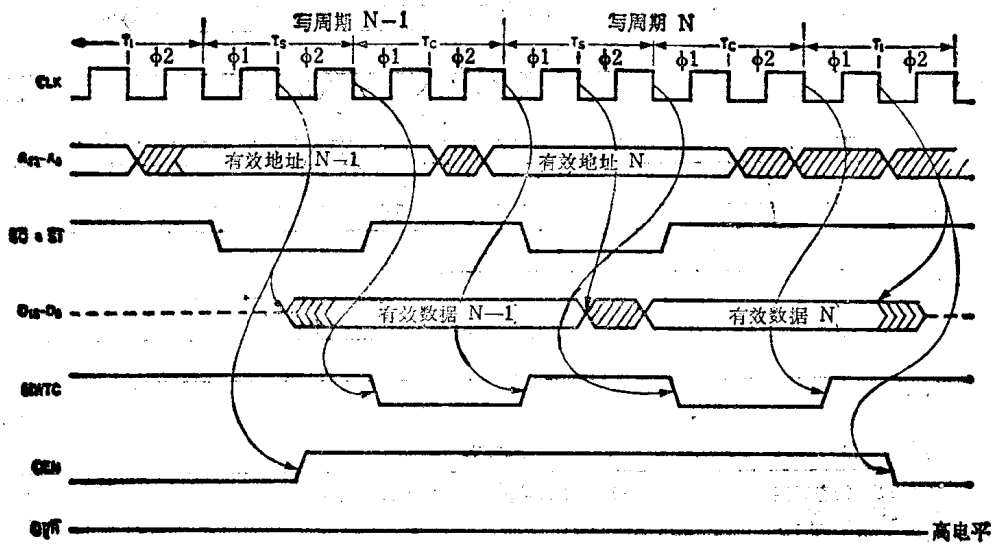


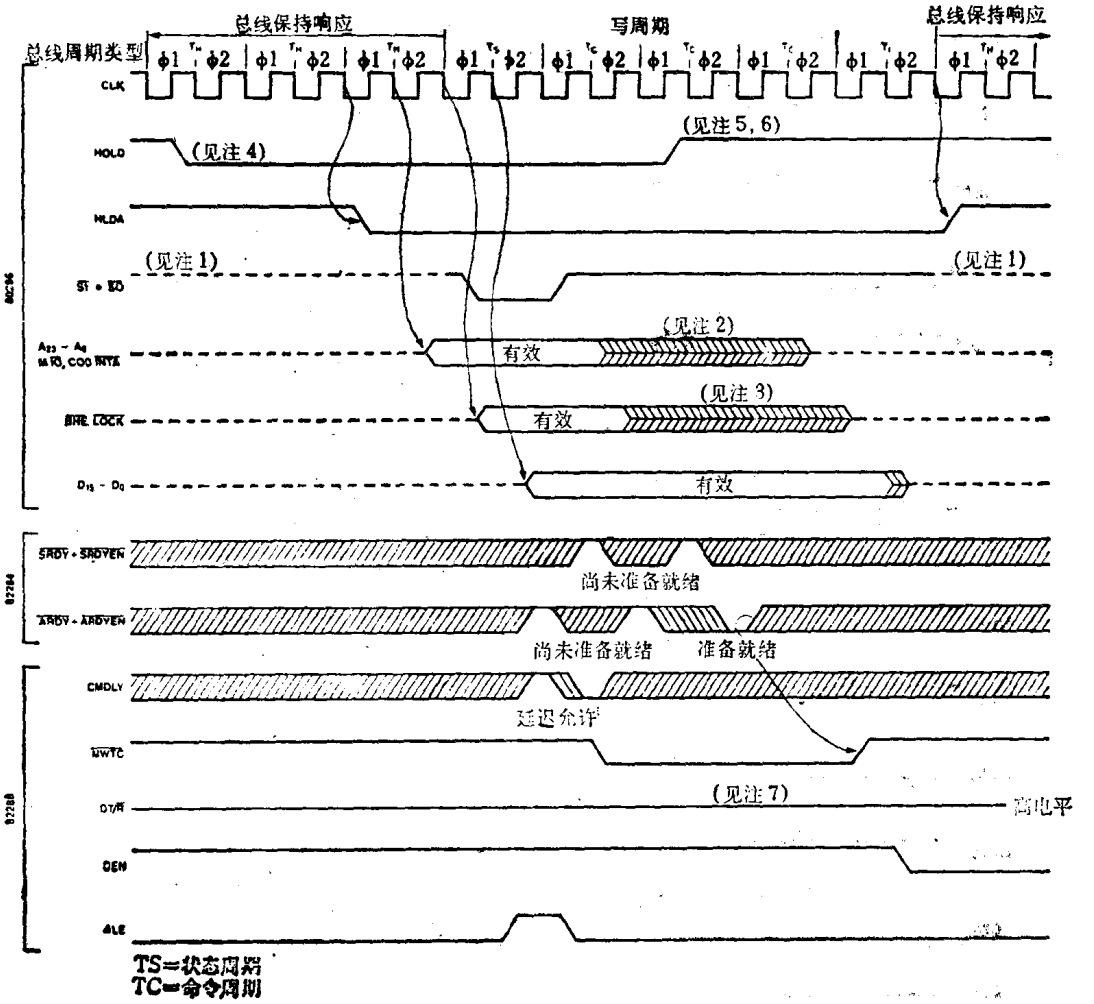
图 15.57 背对背的写—写周期

总线操作进行控制的。对于读操作，DT/$\overline{R}$ 变成低电平。在写操作之前，写操作期间以及两个写操作之间，DT/$\overline{R}$ 保持高电平。

在 T_s 的第 2 个节拍期间，数据总线被写数据信号驱动。在写数据定时中的延迟，使得读数据驱动器在 80286 开始为写数据驱动局部总线之前，有足够的时间从原先的读周期进入第 3 态 OFF。写数据将总是保持有效到超过最后一个 T_c 一个系统时钟，以便为多路总线、或其他类似的存贮器或 I/O 系统，提供足够的保持时间。在写—读或读空闲序列期间，数据总线在最后一个 T_c 之后的处理器周期的第 2 个节拍中，进入第 3 态 OFF。在写—写序列中，数据总线不在 T_c 和 T_s 之间，进入第 3 态 OFF。

§15.6.11 总 线 用 途

80286 局部总线，有如下几种用途：指令数据传送，由其他总线主设备进行的数据传送，取指令，协处理器数据传送，中断响应，暂停/停机。这一节中说明具有专用信号和要求的局部总线的活动。



注：1. 80286 不驱动状态线，但由于 82288 和 82289 中的拉高电阻，状态线在 HOLD 期间保持高电平.

2. 地址、M/$\overline{IO}$ 和 COD/$\overline{INTA}$，在任意 T_0 期间开始浮空，是依赖于 80286 内部的总线仲裁器何时决定把总线释放给外部的 HOLD 请求的. 浮空开始于 T_0 的第 2 节拍.

3 $\overline{BHE}$ 和 LOCK，在任意 T_0 的末尾开始浮空，是依赖于 80286 内部的总线仲裁器决定何时把总线释放给外部的 HOLD 请求的.

4. HOLD↓到 HLDA↓的最小时间如图所示，最大时间比之长一个 T_H.

5. 最早的 HOLD↑ 时间如图所示，它将总是启动一个后读的存贮器周期.

6. 最小的 HOLD↑ 到 HLDA↑ 时间如图所示. 最大时间为指令、总线周期的类型和其他机器状态，如中断、等待、封锁等的函数.

7. 异步准备就绪使周期结束. 在这个例子中，同步准备就绪，并不表示准备就绪. 在准备就绪通过异步输入通知以后，同步准备就绪状态就被忽略.

图 15.58 由异步准备就绪结束多总线写

一、HOLD 和 HLDA

HOLD 和 HLDA，允许其他的总线主设备能通过使 80286 的总线进入 T_H 状态，而获得对局部总线的控制。图 15.58，显示了在 80286 和其他局部总线主设备之间请求转移控制的事件序列。

在这个例子中，80286 开始时由于 HLDA 有效，而处于 T_H 状态。当由 HLDA 变为无效，而离开 T_H 状态时，开始了一个写操作。在该写操作期间，如图中所示，另一个总线主设备利用 HOLD 信号向 80286 请求局部总线。在完成写操作之后，80286 执行一个 T_i 总线周期，以确保写数据保持时间，然后以 HLDA 变成有效，而进入 T_h 状态。

CMDLY 和 $\overline{\text{ARDY}}$ 准备就绪信号，分别用来启动和停止写总线命令。注意，为保证以 $\overline{\text{ARDY}}$ 来结束该周期，$\overline{\text{SRDY}}$ 必须是无效的，或者被 SRDYEN 所禁止。

二、取指令

80286 总线部件(BU)，将在当前指令被执行之前取该指令，这种活动称为预取。它发生在局部总线否则要进入空闲状态的时候，并且服从如下的规则：

(1) 当 6 字节的预取队列中至少有 2 个字节为空时，开始预取操作；

(2) 预取器通常执行字预取，并且这种字预取是不依赖于代码段基地址在物理存贮器中的定位的；

(3) 对把控制转移到一个奇地址的指令，预取器将执行仅取一个字节代码的操作；

(4) 一旦一条控制转移或 HLT 指令被 IU 译码并放入指令队列时，预取就会停止；

(5) 在实地址方式中，预取器可在一个代码段中除最近的控制转移、或 HLT 指令之外取多至 5 个字节；

(6) 在保护方式中，预取器将不会引起段越出异常。预取器会在代码段的最后一个物理存贮器字上停止预取。若程序企图越过代码段的最后一条完整的指令执行，则将引起异常 13；

(7) 若一代码段的最后字节，为偶物理存贮器地址，则预取器将读存贮器的下一物理存贮器字节，以完成一个字代码的取操作。该字节的值将被忽略，任何企图对它的执行，都会引起异常 13。

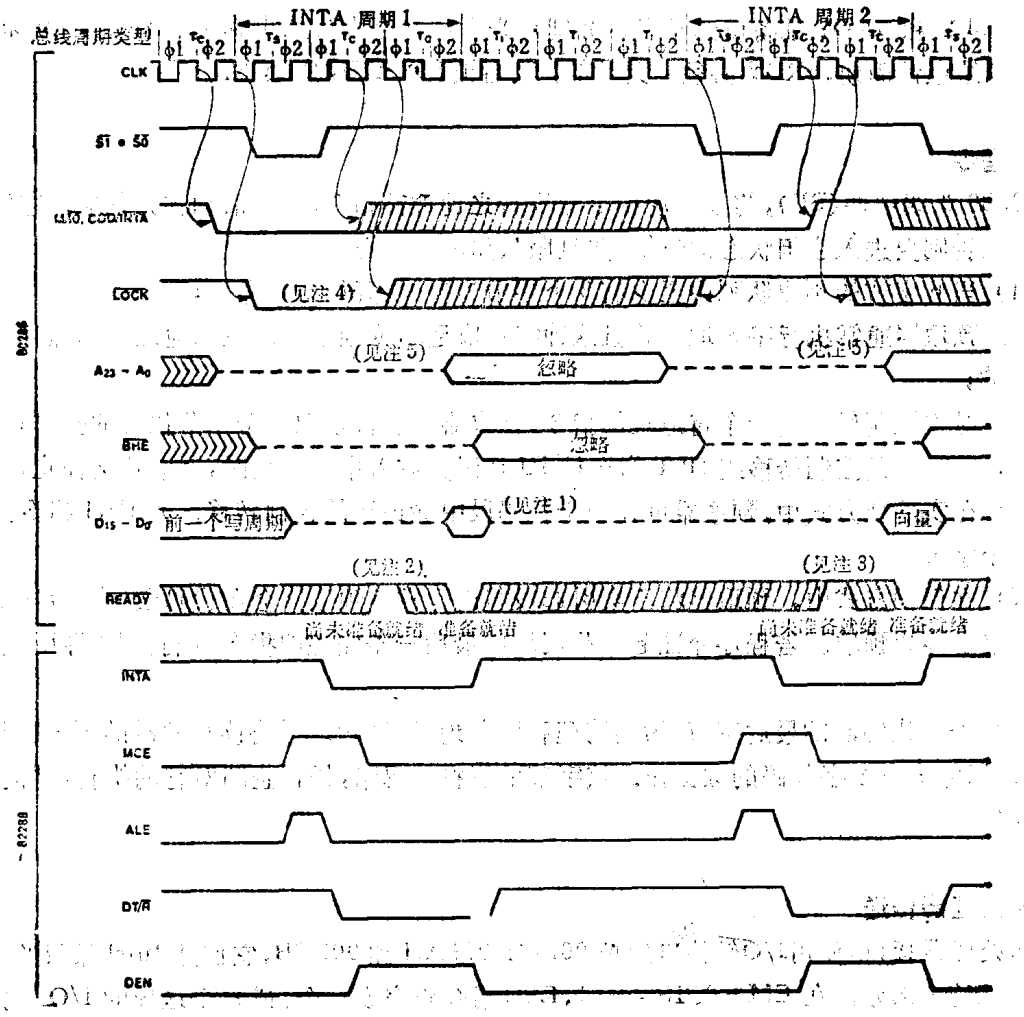
三、协处理器传送

协处理器接口，使用I/O转接口地址 00F8H、00FAH 和 00FCH，它们是 Intel 保留的 I/O 转接口地址的一部分。在 EM=0、TS=0 时，ESC 指令将完成一个或多个这样的 I/O 转接口地址的 I/O 总线操作。这些操作不依赖于 IOPL 和 CPL 的值。

需要访问存贮器的 ESC 指令，能使 CPU 为协处理器的操作数传送而接收 PEREQ 输入。CPU 将决定该操作数的起始地址和指令的读/写状态。对于每个操作数的传送，要执行 2 个或 3 个总线操作，即一个使用 I/O 转接口地址(如 00FAH)的字传送总线操作，和 1 个或 2 个存贮器总线操作。对于每一个定位在奇字节地址的字操作数，需要 3 个总线操作。

四、中断响应序列

80286 在响应 INTR 输入时，所执行的中断响应序列。一个中断响应序列，是由 2 个 INTA 总线操作所组成的。第一个总线操作，使主 8259A 可编程序中断控制器(PIC)、能确定它的从中断控制器中的哪一个将返回中断向量。80286 在第二个总线操作期间，读得一个 8 位的中断向量，用以从中断表中选取一个中断处理程序。在 INTA 总线操作期间(参见图 15.59)，82288 的 MCE (主控制器串联允许)信号，被用来启动串联地址驱动器，把地址放到局部地址总线上，以便通过系统地址总线，分配到从中断控制器中去。在第一个 INTA 总线操作的 T,



注：1. 数据被忽略.

2. 第一个 INTA 周期，必须至少插入一个等待状态，以满足 8259A 的最小 INTA 脉冲宽度.

3. 第二个 INTA 周期，必须至少插入一个等待状态，因为CPU在第一个 T_c 状态之前，将不驱动 A_{23}—A_0、$\overline{BHE}$ 和 $\overline{LOCK}$.

CPU 强加了一个时钟的延迟，以避免由 MCE↓所禁止的串联地址缓冲器和地址输出之间的总线争用.

没有等待状态而在第二个 INTA 周期之后立即开始存贮器周期，则 80286 的地址将是无效的.

4. 在多主设备系统中，LOCK 在第一个 INTA 周期中是有效的，以防止 82289 在 INTA 周期之间释放总线。

5. 在 INTA 周期中的第 2 个 T_c 的节拍 2 期间，A_{23}—A_0 终止处于第 3 态 0FF。

图 15.59　中断响应序列

期间，80286 发出低电平有效的 $\overline{LOCK}$ 信号。在第二个 INTA 总线操作结束之前，局部总线的"保持"请求，将不被响应。

80286 在 INTA 总线操作之间，提供 3 个空闲处理器时钟，以满足最小的 INTA 到 INTA 时间和 8259A 的 CAS(串联地址)输出延迟的需要。第二个 INTA 总线操作，必须至少加上一个通过逻辑控制的 $\overline{READY}$ 的附加 T_c 状态。A_{23}—A_0 在第二个 INTA 总线操作的第一个 T_c 状态之前，一直处于第 3 态 OFF。这样就避免了在串联地址驱动器和 CPU 地址驱动器之间的总线争用。附加的 T_c，给 80286 提供了使之能重新启动地址总线，去完成后续的总线操作的时间。

五、局部总线使用优先权

80286 局部总线，是由几个内部部件和外部的 HOLD 请求共享的。在同时请求的情况下，它们的优先权为：

(最高) (1) 任何发出 $\overline{LOCK}$ 的传送，其中 $\overline{LOCK}$ 可以是显式的(通过 LOCK 指令前缀)，也可以是隐式的(如段描述子访问、中断响应序列或涉及存贮器的 XCHG)。

(2) 奇地址定位的字操作数所需要的双字节总线操作的第二个总线操作。

(3) 通过 HOLD 输入，请求局部总线。

(4) 通过 PEREQ 输入的协处理器数据操作数传送。

(5) 作为指令的一部分，而由 EU 完成的数据传送。

(最低) (6) 来自 BU 的指令预取请求。EU 将提前 2 个处理器时钟禁止预取，以使 EU 等待一次预取完成的时间最少。

六、暂停或停机周期

在总线操作时，80286 在外部指出暂停或停机的情况。这些情况的产生，是由于执行 HLT 指令、或在企图执行一条指令时产生多个保护异常。暂停或停机总线操作是由 $\overline{S_1}$、$\overline{S_0}$ 和 COD/$\overline{INTA}$ 为低电平而 M/$\overline{IO}$ 为高电平来标识的。A_1 为高电平表示出现了暂停，A_1 为低电平，则表示出现了停机。82288 总线控制器将不发出结束一个暂停、或停机总线操作的 ALE 和 $\overline{READY}$。

在暂停或停机期间，80286 可以为 PEREQ、或 HOLD 请求服务。在停机期间，协处理器段越出异常将禁止进一步的 PEREQ 服务。NMI 或 RESET，将迫使 80286 退出暂停或停机状态。若中断是允许的，则 INTR 或协处理器段越出异常也将迫使 80286 退出暂停。

§15.7 系统结构

iAPX286 微系统多用途的总线结构，具有一整套的支持芯片，从而使系统在大范围内有灵活的结构。图 15.60 所示的 iAPX286 基本结构类似于 iAPX86 的最大方式系统。它包括 CPU 以及一个 8259A 中断控制器，一个 82284 时钟发生器，一个 82288 总线控制器。iAPX86 的锁存器 8282 和 8283 以及收发器 8286 和 8287 可以应用于 iAPX286 微系统中。

如图 15.60 中虚线部分所示的那样，可以加接协处理器是 iAPX286 微系统的一个主要特点。协处理器接口，使外部硬件能在 CPU 执行其他指令的同时，完成专用的功能和传送数

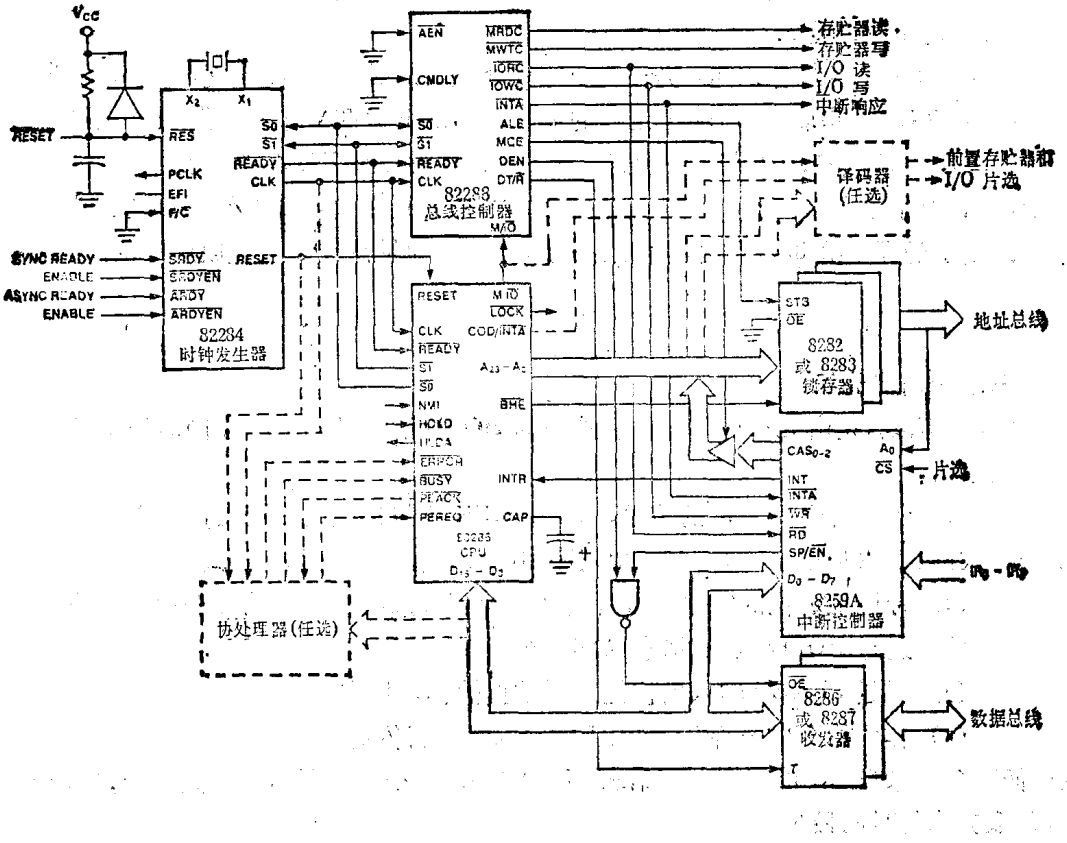


图 15.60　基本 iAPX286 系统结构

据。由于 80286 监督着协处理器所有的数据传送和指令执行，所以整个系统仍是一个整体。

包含有 80287 数值协处理器 (NPX) 的 iAPX286/20 数值数据处理器使用这样的接口，iAPX286/20 具有 iAPX86/20 或 iAPX88/20 所有的指令和数据类型。80287NPX 能在 CPU 程序运行的同时，完成数值计算和数据传送。数值代码和数据，具有与被 iAPX286 保护机构所保护的所有其他信息相同的完整性。

在前一个总线操作的数据传送期间，80286 能重叠片选译码和地址传播。这个信息由 ALE 在 T_2 周期的中部锁存在 8282/3 中，当下一周期的地址正在被译码、并传播到系统中去时，锁存的片选和地址信号，在总线操作期间仍保持稳定。译码逻辑，可以用一个高速的双极型 PROM 来实现。图 15.60 中所示的译码逻辑，利用 80286 总线周期的地址和数据之间重叠的优点，来产生提前的存贮器和 I/O 选择信号。这样可使由地址传播和译码延迟所引起的系统性能的下降减至最小。除了选择存贮器和 I/O 之外，这种预选还用在支持局部和系统总线的结构中，来为每个总线周期启动相应的总线接口。译码逻辑中使用 COD/$\overline{\text{INTA}}$ 和 M/$\overline{\text{IO}}$ 信号来识别中断、I/O、代码和数据总线周期。

通过增加 82289 总线仲裁器，80286 提供了如图 15.61 所示的多总线系统总线接口。用于多总线的 82288ALE 输出，连接到它本身的 CMDY 输入上，从而把命令的开始延迟一个系统 CLK，以满足多总线地址和写数据建立时间的要求。这样的安排，至少将为每个用于多总线的

总线操作，增加一个附加的 T_c 状态。

辅助的 82288 总线控制器和附加的锁存器以及收发器，可以加到如图 15.61 所示的局部总线上。这种结构使 80286 能支持用于局部存贮器和外围设备的在板总线和用于系统总线接口的多总线。

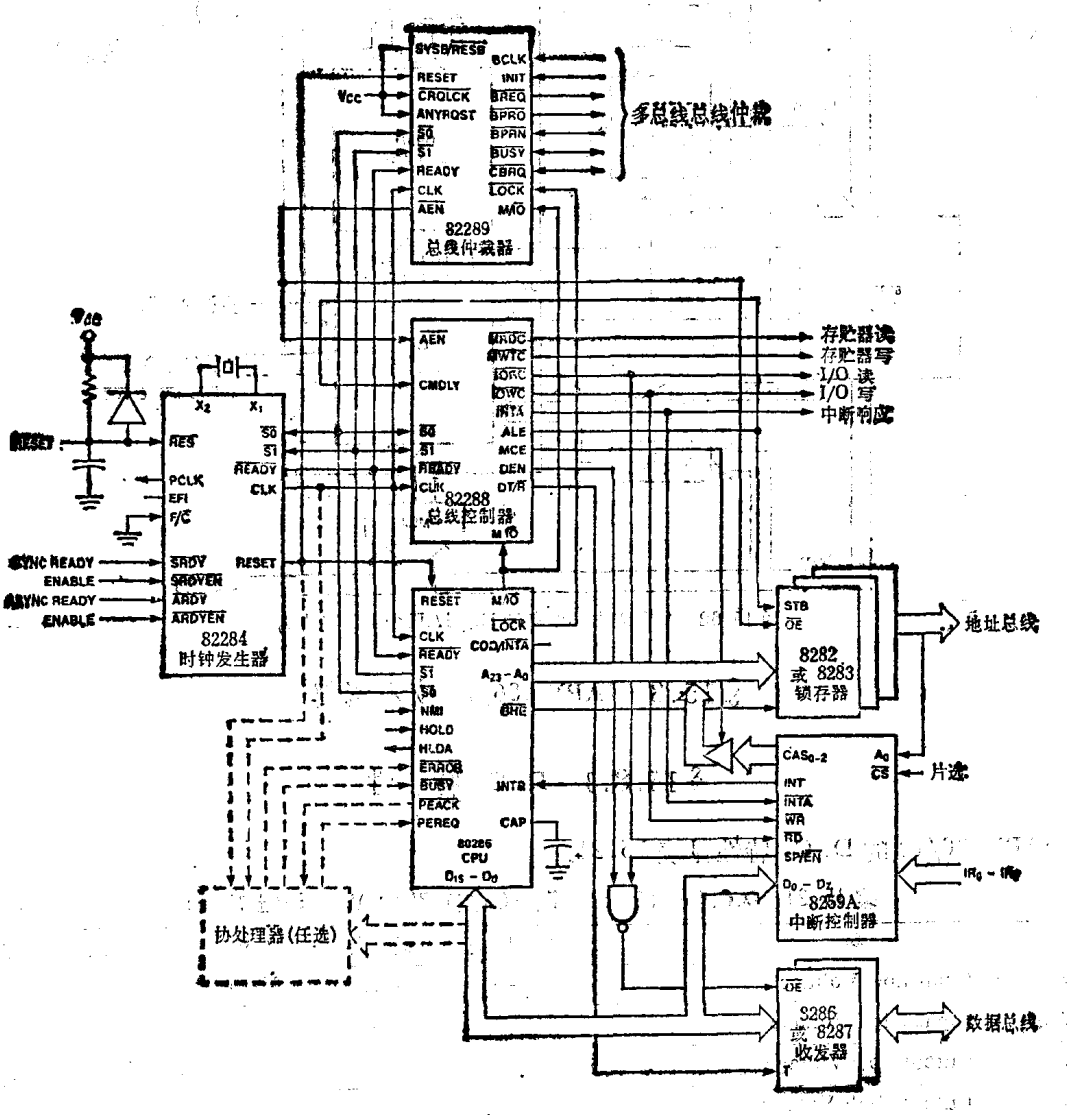


图 15.61 多总线系统总线接口

图 15.62，表示了在多总线系统总线和 iAPX 286 局部总线之间，加上了双向端口动态存贮器。双向端口接口，是由 8207 双向端口 DRAM 控制器提供的。8207 与 CPU 同步运行，以使局部存贮器引用的吞吐量达到最大。它还仲裁来自局部和系统总线的请求，并完成诸如刷新、RAM 的初始化和读/修改/写周期等功能。8207 与 8206 错误检测和纠正存贮器控制器相结合，提供了纠单错的能力。双向端口存贮器，可以与标准的多总线系统总线接口相结合，以使多处理器系统结构的性能和保护，达到最高的指标。

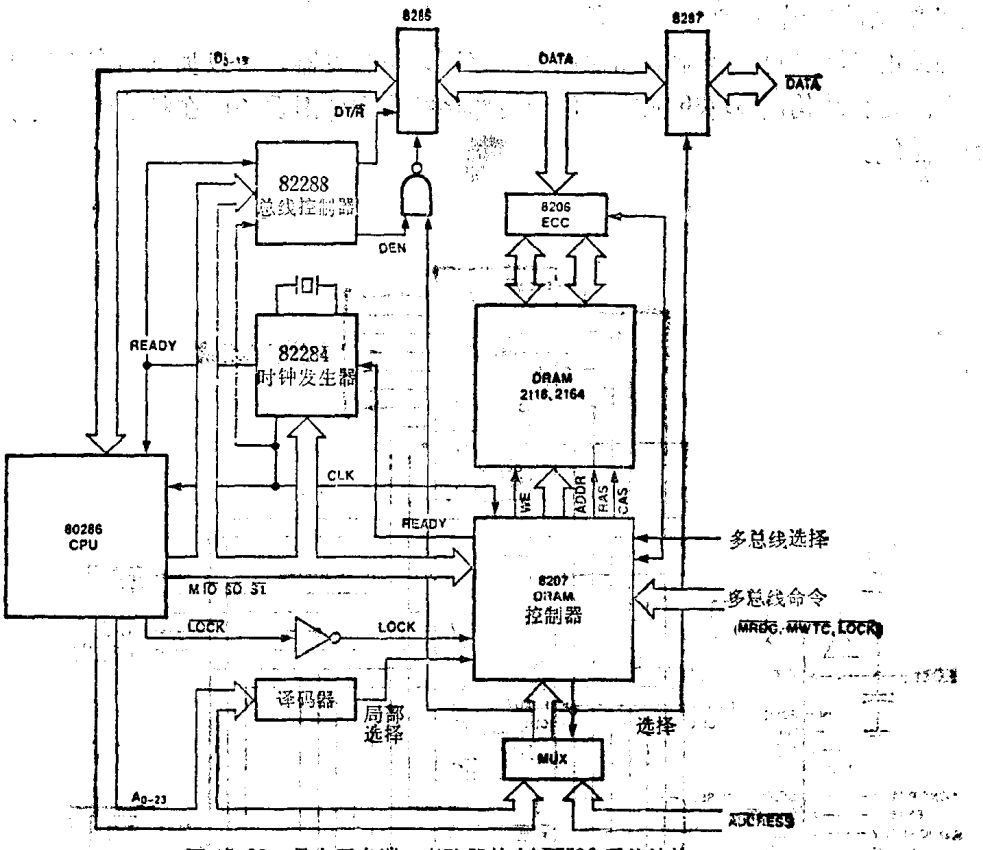


图 15.62　具有双向端口存贮器的 iAPX86 系统结构

§15.8　iAPX 286/10 技术资料

§15.8.1　D.C. 特　性

iAPX286/10 的 D. C. 特性见表 15.21。

表 15.21　D.C.特性(80286: T_A = 0°C 到 70°C V_{cc} = 5V ± 10%)

符　号	参　数	最　小	最　大	单　位	测　试　条　件
V_{IL}	Input Low Voltage	−0.5	+0.8	V	
V_{IH}	Input High Voltage	2.0	V_{cc}+0.5	V	
V_{OL}	Output Low Voltage		0.45	V	I_{OL}=3.0mA
V_{OH}	Output High Voltage	2.4		V	I_{OH}=−400μA
I_{CC}	Power Supply Current		600	mA	T_A=25℃
I_{LI}	Input Leakage Current		±10	μA	0V≤V_{IN}≤V_{CC}
I_{LO}	Output Leakage Current		±10	μA	0.45V≤V_{OUT}≤V_{CC}
V_{CL}	Clock Input Low Voltage	−0.5	+0.6	V	
V_{CH}	Clock Input High Voltage	3.8	V_{CC}+1.0	V	
C_{IN}	Capacitance of Inputs (All input except CLK)		10	PF	fc=1MHz
C_O	Capacitance of I/O or outputs		20	PF	fc=1MHz
C_{CLK}	Capacitance of CLK Input		12	PF	fc=1MHz

§15.8.2 A.C. 特 性

iAPX286/10 的 A. C. 特性见表 15.22。

表 15.22 A. C. 特性(T_A = 0°C 到 70°C, V_{cc} = 5V ± 10%)

(a) 80286 需要的定时

符 号	参 数	最 小	最 大	单 位	测 试 条 件
1	System clock period	62.5	250	ns	
2	System clock low time	15	230	ns	at 6Volts
3	System clock high time	20	235	ns	at 3.2Volts
4	Asynchronous input setup time	20		ns	See note 1
5	Asynchronous input hold time	20		ns	See note 1
6	RESET setup time	20		ns	
7	RESET hold time	0		ns	
8	Read data in setup time	10		ns	
9	Read data in hold time	5		ns	
10	READY setup time	28.5		ns	
11	READY hold time	25		ns	
12	STATUS/PEACK valid delay	0	40	ns	
13	Address valid delay	0	60	ns	
14	Write data valid delay	0	50	ns	C_L=100Pfd max
15	Address/Status/Data float delay	0	60	ns	
16	HLDA valid delay	0	60	ns	

(b) 82284 需要的定时

符 号	参 数	最 小	最 大	单 位	测 试 条 件
17	$\overline{\text{SRDY}}$/$\overline{\text{SRDYEN}}$ setup time	15		ns	
18	$\overline{\text{SRDY}}$/$\overline{\text{SRDYEN}}$ hold time	0		ns	
19	$\overline{\text{ARDY}}$/$\overline{\text{ARDYEN}}$ setup time	−5		ns	See note 1 见注 1
20	$\overline{\text{ARDY}}$/$\overline{\text{ARDYEN}}$ hold time	16		ns	See note 1 见注 1
21	PCLK delay	0	40	ns	C_L=75pfd I_{OL}=5.25mA I_{OH}=−1.05mA

注 1：这些时间是为测试而给出的，以确保顶定的动作。

(c) 82288 需要的定时

符号	参数	最小	最大	单位	测试条件
22	CMDLY setup time	20		ns	
23	CMDLY hold time	0		ns	
24	Command delay	5	25	ns	C_L=300pfd max I_{OL}=32mA max I_{OH}=−5mA max
25	ALE active delay	2.5	12	ns	
26	ALE inactive delay	0	15	ns	
27	DTR read active delay	0	25	ns	C_L=80pfd max I_{OL}=16mA max I_{OH}=−1mA max
28	DTR read inactive delay	15	40	ns	
29	DEN read active delay	10	50	ns	
30	DEN read inactive delay	2.5	20	ns	
31	DEN write active delay	17.5	40	ns	
32	DEN write inactive delay	12.5	47.5	ns	

§15.8.3 波　形

图 15.63—15.68 是波形图，其中定时可参见表 15.21 和 15.22。

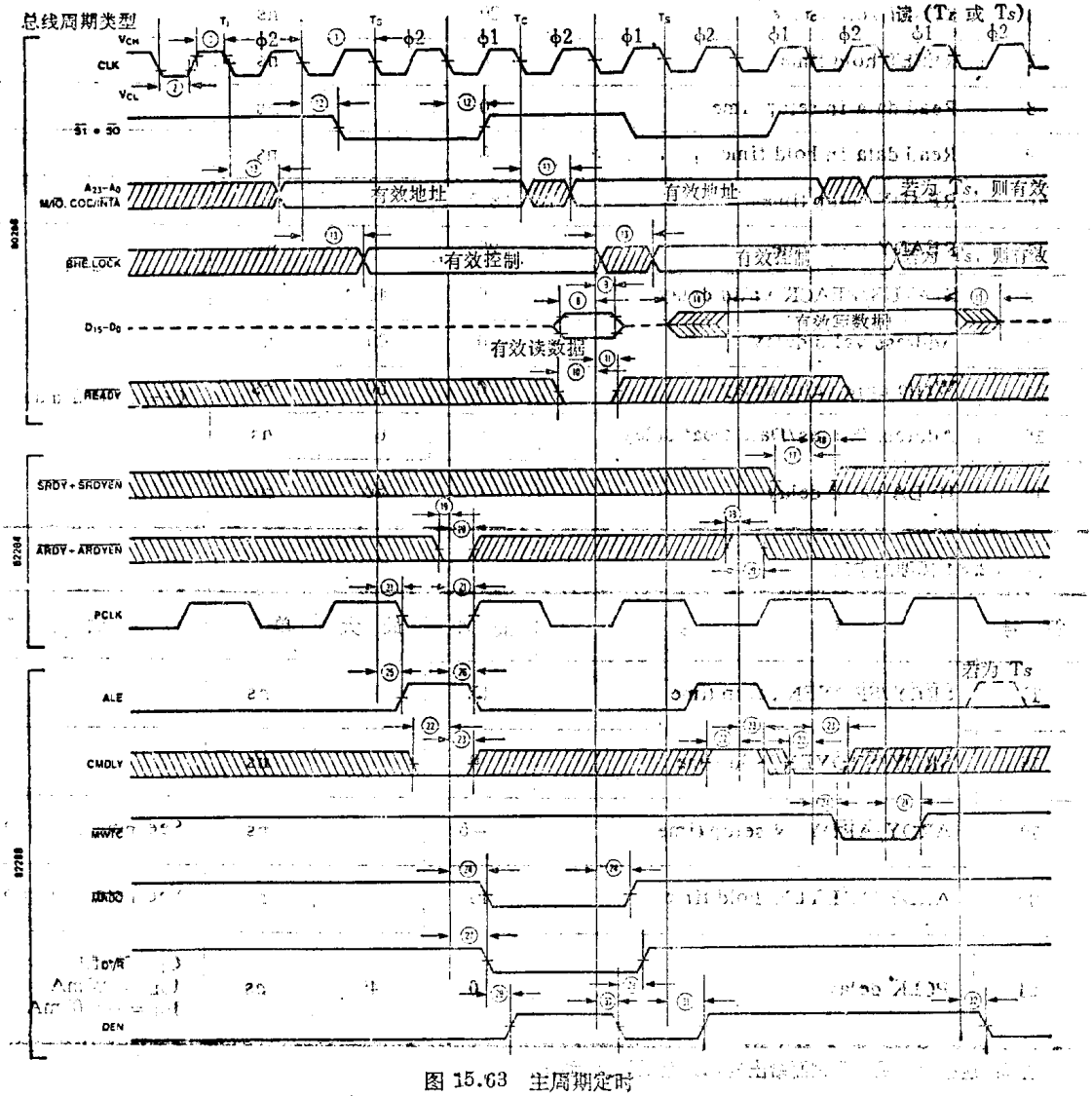
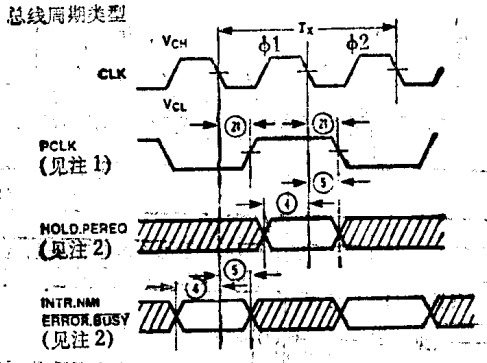


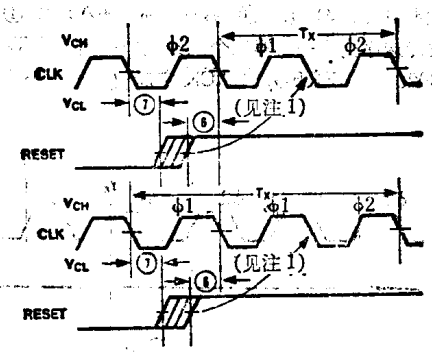
图 15.63　主周期定时



注：1. PCLK 指示在下一个 CLK，将出现处理器周期的哪个节拍．在第一个总线周期被执行之前，PCLK 不能指示正确的节拍．

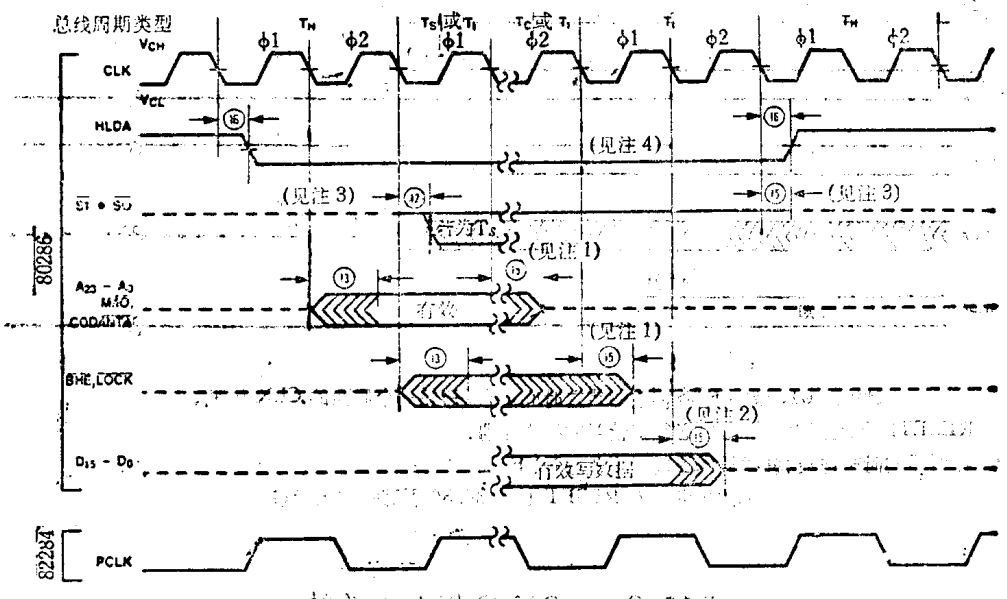
2 这些输入是异步的．所示的建立和保持时间能确保以测试为目的的识别．

图 15.64 80286 异步输入信号定时



注：当 RESET 与所示的建立时间相遇时，下一个 CLK 将启动或重复一个处理器周期的节拍 1.

图 15.65 80286 RESET 输入定时和后续处理器周期节拍



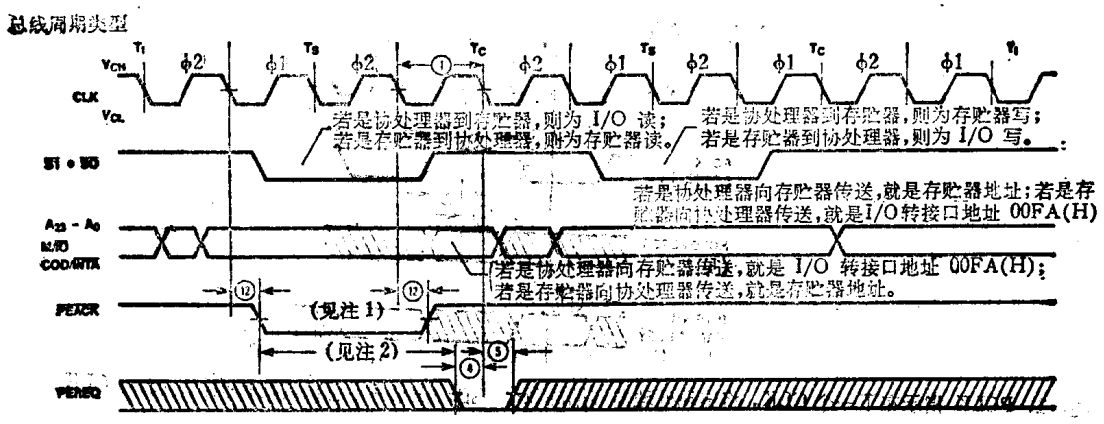
注：1. 这些信号还可以在所示的时间内，被 80286 驱动．依据最新浮空时间的最坏情况，已在图中表示．

2 若在图中 T_1 之前，最近的周期是一个写 T_c，则数据总线将被如所示的那样驱动．

3. 80286 在 T_H 期间浮空它的状态引脚．外部拉高电阻(在 82288)中，使这些信号保持高电平．

4. 对 HOLD 请求建立 HLDA，参见图 15.58.

图 15.66 输入和终止 HOLD

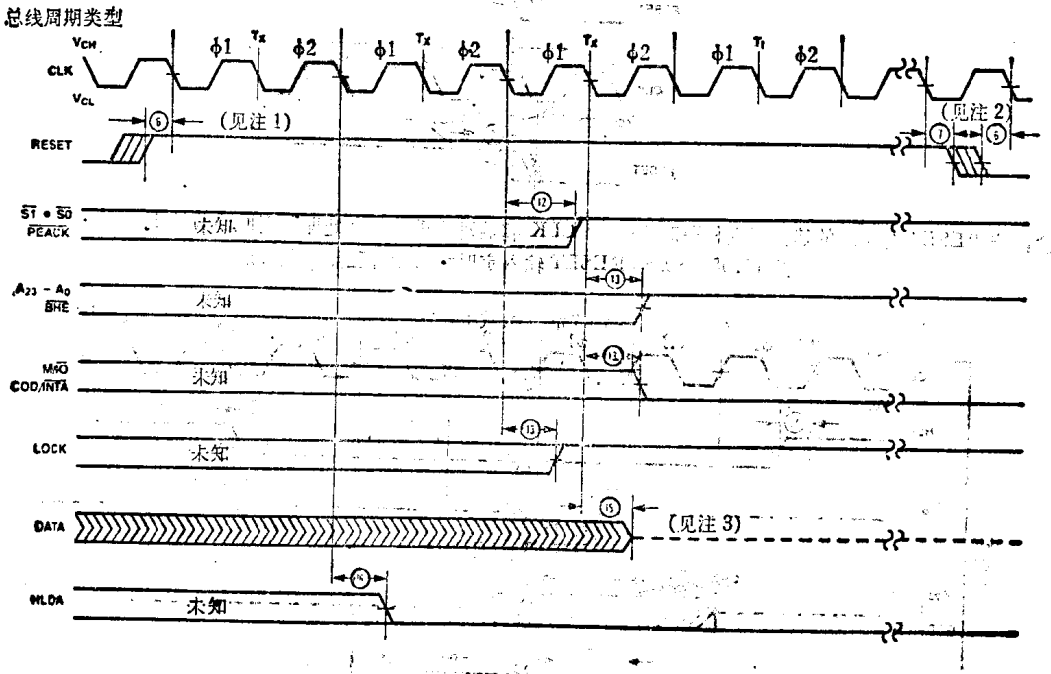


注：1. 在一个协处理器数据操作数传送序列的第一个总线操作期间，PEACK 总是变得有效。

2. 为了防止第二个协处理器数据操作数传送，最坏情况的最大时间(如上所示)是：3×①—⑪max—④min。事实上，依赖于结构的最大时间是：3×①—⑪max—④min+A×2×①。

A 是加给协处理器数据操作数传送序列第一或第二个总线操作的附加的 T_c 状态的个数。

图 15.67 80286 的 PEREQ/PEACK 定时仅需要一个传送的 PEREQ 定时



注：1. 用处理器时钟的 $\phi 1$，破坏 RESET↑ 的建立时间，可以开始一个后面的 CLK 周期。

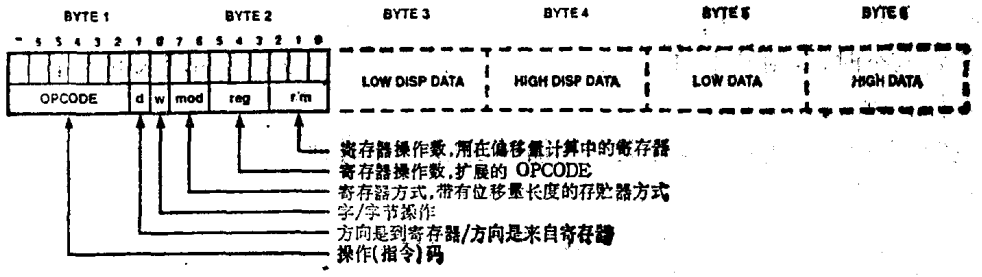
2. RESET↓ 的建立和保持时间，必须与恰当的操作相遇。

3. 在所示的时刻，数据总线只保证处于第 3 态 OFF。

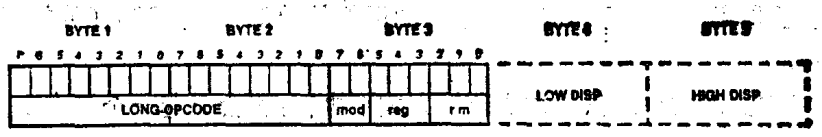
图 15.68 在 RESET 期间，80286 引脚的初始状态

§15.8.4 80286 指令集总结

图 15.69，是 80286 指令格式的例子。从图中可见，80286 的指令格式，可以分成短操作码和长操作码两类，最长的指令可以有 6 字节长。



A. 短操作码格式的例子



B. 长操作码格式的例子

图 15.69 80286 指令格式的例子

一、指令定时注释

下面所列的指令时钟计数，确定了 80286 最大的执行速率。在总线周期没有延迟的情况下，80286 程序的真实时钟计数，将比计算所得的时钟计数平均多 5%，这是因为指令序列执行起来，比从存贮器中把它们取出来要快。

为了计算执行指令序列所用去的时间，可以如下面表中所示的那样，用处理器时钟周期乘以所有指令时钟计数的和。8MHz 的处理器时钟的时钟周期是 125 毫微秒，它需要 16MHz 的 80286 系统时钟(CLK 输入)。

二、关于指令时钟计数的假定

(1) 指令已被预取出，译码好，并已准备就绪可以执行。控制转移指令的时钟计数包括需要用来预取、译码和为执行而准备下条指令的所有时间。

(2) 总线周期不需要等待状态。

(3) 没有协处理器数据传送或局部总线的 HOLD 请求。

(4) 在指令执行时不出现异常。

三、指令集总结注释

由 MOD 字段选择的地址位移量没有表示出来。若是必须的，它们出现在所示的指令字段的后面。

高于/低于是对无符号值而言的。

大于是对正的带符号值而言的。

小于是对更小的正的(大的负的)带符号值而言的。

若 d = 1，则送到寄存器；若 d = 0 则来自寄存器。

若 w = 1，则为字指令；若 w = 0 则为字节指令。

若 s = 0，则 16 位立即数据形成操作数。

若 s = 1，则一个立即数据字节，要被符号扩展，以形成 16 位操作数。

X 表示忽略。

Z 串基本指令，用来与 ZF 标志进行比较。

若给定两个时钟，则较小的一个数，是对应于寄存器操作数的；而较大的一个数，是对应于存贮器操作数的。

* = 在计算偏移量需要把 3 个元素相加时，加上一个时钟。

n = 重复的次数。

m = 在下条指令中的代码的字节数。

Level(L)——过程的字典嵌套层。

下面的注释描述了 80286 两种操作方式下指令的可能的异常、副作用和许可的用途。

仅对实地址方式：

1. 这是一条保护方式的指令。试图在实地址方式下执行，将会引起一个没有定义的操作码异常(6)。

2. 若在偏移量 0FFFFH 引用一个字操作数，则将会引起一个段越出异常。

3. 这条指令可以在实地址方式下执行，以便为 CPU 的保护方式进行初始化。

4. IOPL 和 NT 字段将保持 0。

5. 若操作数超过了段限，则将引起协处理器段越出中断(9)。

对任一种方式：

6. 依据操作数的值，能引起一个异常。

7. 不管是否出现 LOCK 指令前缀，$\overline{\text{LOCK}}$ 信号将自动地发出。

仅对保护虚地址方式：

8. INT、JMP、CALL、RET 或 IRET 指令的目，必须在一个代码段的限内，否则将会引起通用保护异常(13)。

9. 若由于违反段限或访问权，而使存贮器操作数不能使用，则将引起一个通用保护异常(13)。若违反堆栈段的限，则将引起堆栈段越出异常(12)。

10. 对段装入操作，CPL、RPL 和 DPL 必须与特权规则一致，以避免出现异常。该段必须存在，以避免不存在异常(11)。若 SS 寄存器是目，并且出现了段不存在情况，则将引起堆栈异常(12)。

11. 由这条指令在 GDT 或 LDT 中所作的所有段描述子的访问，将自动地发出 $\overline{\text{LOCK}}$，以便在多处理器系统中，保持描述子的完整性。

12. 引用另一个代码段的 JMP、CALL、INT、RET、IRET 指令，若违反特权规则，则将引起通用保护异常(13)。

13. 若 CPL≠0，则将引起通用保护异常(13)。

14. 若 CPL > IOPL，则将引起通用保护异常(13)。

15. 若 CPL > IOPL，则标志字的 IF 字段不被更新。IOPL 字段只在 CPL = 0 时才被更新。

16. 把任何对特权规则的违例，应用于选择子操作数，不导致保护异常；然而不返回结果，并且 0 标志被清除。

17. 若存贮器操作数的起始地址违反了段限，或试图进行无效的访问，则在该 ESC 指令被执行之前，将会引起一个通用保护异常(13)。若被操作数的起始地址超越了堆栈段的限，则将引起一个堆栈段越出异常。若在试图进行数据传送时违反了段限，则会引起协处理器段越出异常(9)。

表 15.23 80286 指令集

功能	格式	时钟计数		注释	
		实地址方式	保护虚地址方式	实地址方式	保护虚地址方式
数据传送					
MOV=Move:					
寄存器到寄存器/存贮器	1 0 0 0 1 0 0 w | mod reg r/m	2,3*	2,3*	2	9
寄存器/存贮器到寄存器	1 0 0 0 1 0 1 w | mod reg r/m	2,5*	2,5*	2	9
立即数到寄存器/存贮器	1 1 0 0 0 1 1 w | mod 000 r/m | data | data if w=1	2,3*	2,3*	2	9
立即数到寄存器	1 0 1 1 w reg | data | data if w=1	2	2		
存贮器到累加器	1 0 1 0 0 0 0 w | addr-low | addr-high	5	5	2	9
累加器到存贮器	1 0 1 0 0 0 1 w | addr-low | addr-high	3	3	2	9
寄存器/存贮器到段寄存器	1 0 0 0 1 1 1 0 | mod 0 reg r/m	2,5*	17,19*	2	9,10,11
段寄存器到寄存器/存贮器	1 0 0 0 1 1 0 0 | mod 0 reg r/m	2,3*	2,3*	2	9
PUSH=Push					
存贮器	1 1 1 1 1 1 1 1 | mod 110 r/m	5*	5*	2	9
寄存器	0 1 0 1 0 reg	3	3	2	9
段寄存器	0 0 0 reg 1 1 0	3	3	2	9
立即数	0 1 1 0 1 0 s 0 | data | data if s=0	3	3	2	9
PUSHA=推入所有寄存器	0 1 1 0 0 0 0 0	17	17	2	9
POP=Pop:					
存贮器	1 0 0 0 1 1 1 1 | mod 000 r/m	5*	5*	2	9
寄存器	0 1 0 1 1 reg	5	5	2	9
段寄存器	0 0 0 reg 1 1 1 (reg≠01)	5	20	2	9,10,11

续 表

功 能	格 式		时钟计数		注 释	
			实地址方式	保护虚地址方式	实地址方式	保护虚地址方式
POPA=弹出所有寄存器	01100001		19	19	2	9
XCHG=Exchange						
寄存器/存贮器与寄存器	1000011w mod reg r/m		3,5*	3,5*	2,7	7,9
寄存器与累加器	10010 reg		3	3		
IN=Input from:						
固定的转接口	1110010w port		5	5		14
可变的转接口	1110110w		5	5		14
OUT=Output to:						
固定的转接口	1110011w port		3	3		14
可变的转接口	1110111w		3	3		14
XLAT=翻译字节送 AL	11010111		5	5		9
LEA=把 EA 装入寄存器	10001101 mod reg r/m		3*	3*		
LDS=把指示器装入 DS	11000101 mod reg r/m	(mod≠11)	7*	21*	2	9,10,11
LES=把指示器装入 ES	11000100 mod reg r/m	(mod≠11)	7*	21*	2	9,10,11
LAHF=把标志装入 AH	10011111		2	2		
SAHF=把 AH 存放到标志	10011110		2	2		
PUSHF=推入标志	10011100		3	3	2	9
POPF=弹出标志	10011101		5	5	2,4	9,15

续 表

功 能	格 式	时钟计数		注 释	
		实地址方式	保护虚地址方式	实地址方式	保护虚地址方式
算术					
ADD=Add:					
寄存器/存贮器与寄存器，结果送目操作数	0 0 0 0 0 0 d w | mod reg r/m	2,7*	2,7*	2	9
立即数与寄存器/存贮器，结果送寄存器/存贮器	1 0 0 0 0 0 s w | mod 000 r/m | data | data if s w=01	3,7*	3,7*	2	9
立即数与累加器，结果送累加器	0 0 0 0 0 1 0 w | data | data if w=1	3	3		
ADC=Add with carry:					
寄存器/存贮器与寄存器，结果送目操作数	0 0 0 1 0 0 d w | mod reg r/m	2,7*	2,7*	2	9
立即数与寄存器/存贮器，结果送寄存器/存贮器	1 0 0 0 0 0 s w | mod 010 r/m | data | data if s w=01	3,7*	3,7*	2	9
立即数与累加器，结果送累加器	0 0 0 1 0 1 0 w | data | data if w=1	3	3		
INC=Increment:					
寄存器/存贮器	1 1 1 1 1 1 1 w | mod 000 r/m	2,7*	2,7*	2	9
寄存器	0 1 0 0 0 reg	2	2		
SUB=Subtract:					
寄存器/存贮器和寄存器，结果送目操作数	0 0 1 0 1 0 d w | mod reg r/m	2,7*	2,7*	2	9
从寄存器/存贮器中减去立即数	1 0 0 0 0 0 s w | mod 101 r/m | data | data if s w=01	3,7*	3,7*	2	9
从累加器中减去立即数	0 0 1 0 1 1 0 w | data | data if w=1	3	3		
SBB=Subtract with borrow:					
寄存器/存贮器和寄存器，结果送目操作数	0 0 0 1 1 0 d w | mod reg r/m	2,7*	2,7*	2	9
从寄存器/存贮器中减去立即数	1 0 0 0 0 0 s w | mod 011 r/m | data | data if s w=01	3,7*	3,7*	2	9
从累加器中减去立即数	0 0 0 1 1 1 0 w | data | data if w=1	3	3		

续 表

功 能	格 式	时钟计数		注 释	
		实地址方式	保护虚地址方式	实地址方式	保护虚地址方式
DEC=Decrement;					
寄存器/存贮器	1 1 1 1 1 1 1 w | mod 001 r/m	2,7*	2,7*	2	9
寄存器	0 1 0 0 1 reg	2	2		
CMP=Compare;					
寄存器/存贮器与寄存器	0 0 1 1 1 0 1 w | mod reg r/m	2,6*	2,6*	2	9
寄存器与寄存器/存贮器	0 0 1 1 1 0 0 w | mod reg r/m	2,7*	2,7*	2	9
立即数与寄存器/存贮器	1 0 0 0 0 0 s w | mod 111 r/m | data | data if s w=01	3,6*	3,6*	2	9
立即数与累加器	0 0 1 1 1 1 0 w | data | data if w=1	3	3		
NEG=改变符号	1 1 1 1 0 1 1 w | mod 011 r/m	2	7*	2	9
AAA=ASCII 加调整	0 0 1 1 0 1 1 1	3	3		
DAA=十进制加调整	0 0 1 0 0 1 1 1	3	3		
AAS=ASCII 减调整	0 0 1 1 1 1 1 1	3	3		
DAS=十进制减调整	0 0 1 0 1 1 1 1	3	3		
MUL=乘(无符号);	1 1 1 1 0 1 1 w | mod 100 r/m				
寄存器一字节		13	13		
寄存器一字		21	21		
存贮器一字节		16*	16*	2	9
存贮器一字		24*	24*	2	9
IMUL=整数乘(带符号);	1 1 1 1 0 1 1 w | mod 101 r/m				
寄存器一字节		13	13		
寄存器一字		21	21		

续 表

功 能	格 式	时钟计数		注 释	
		实地址方式	保护虚地址方式	实地址方式	保护虚地址方式
存贮器一字节					
存贮器一字					
IMUL=整数立即数乘(带符号)	0 1 1 0 1 0 s 1 | mod reg r/m | data | data if s=0	21,24*	21,24*	2	9
DIV=Divide(无符号):	1 1 1 1 0 1 1 w | mod 110 r/m				
寄存器一字节		14	14		
寄存器一字		22	23		
存贮器一字节		17*	17*	2,6	6,9
存贮器一字		25*	25*	2,6	6,9
算术(续)					
IDIV=整数除(带符号):	1 1 1 1 0 1 1 w | mod 111 r/m				
寄存器一字节		17	17		
寄存器一字		25	25		
存贮器一字节		20*	20*	2	9
存贮器一字		28*	28*	2	9
AAM=ASCII 乘调整	1 1 0 1 0 1 0 0 | 0 0 0 0 1 0 1 0	16	16		
AAD=ASCII 除调整	1 1 0 1 0 1 0 1 | 0 0 0 0 1 0 1 0	14	14		
CBW=把字节转换为字	1 0 0 1 1 0 0 0	2	2		
CWD=把字转换为双字	1 0 0 1 1 0 0 1	2	2		
逻辑					
移位/环移指令:					
寄存器/存贮器移1位	1 1 0 1 0 0 0 w | mod TTT r/m	2,7*	2,7*	2	9
寄存器/存贮器移 CL 位	1 1 0 1 0 0 1 w | mod TTT r/m	5+n,8+n*	5+n,8+n*	2	9
寄存器/存贮器移 Count 位	1 1 0 0 0 0 0 w | mod TTT r/m | count	5+n,8+n*	5+n,8+n*	2	9

续 表

功 能	格 式	时钟计数		注 释	
		实地址方式	保护虚地址方式	实地址方式	保护虚地址方式
	TTT 指令 0 0 0 ROL 0 0 1 ROR 0 1 0 RCL 0 1 1 RCR 1 0 0 SHL/SAL 1 0 1 SHR 1 1 1 SAR				
AND=And; 寄存器/存贮器与寄存器，结果送目操作数	0 0 1 0 0 0 d w | mod reg r/m	2,7*	2,7*	2	9
立即数与寄存器/存贮器，结果送寄存器/存贮器	1 0 0 0 0 0 0 w | mod 100 r/m | data | data if w=1	3,7*	3,7*	2	9
立即数与累加器，结果送累加器	0 0 1 0 0 1 0 w | data | data if w=1	3	3		
TEST=And function to flags, no result; 寄存器/存贮器与寄存器	1 0 0 0 0 1 0 w | mod reg r/m	2,6*	2,6*	2	9
立即数据与寄存器/存贮器	1 1 1 1 0 1 1 w | mod 000 r/m | data | data if w=1	3,6*	3,6*	2	9
立即数据与累加器	1 0 1 0 1 0 0 w | data | data if w=1	3	3		
OR=Or 寄存器/存贮器与寄存器，结果送目操作数	0 0 0 0 1 0 d w | mod reg r/m	2,7*	2,7*	2	9
立即数与寄存器/存贮器，结果送寄存器/存贮器	1 0 0 0 0 0 0 w | mod 001 r/m | data | data if w=1	3,7*	3,7*	2	9
立即数与累加器，结果送累加器	0 0 0 0 1 1 0 w | data | data if w=1	3	3		
XOR=Exclusive or; 寄存器/存贮器与寄存器，结果送目操作数	0 0 1 1 0 0 d w | mod reg r/m	2,7*	2,7*	2	9
立即数与寄存器/存贮器，结果送寄存器/存贮器	1 0 0 0 0 0 0 w | mod 110 r/m | data | data if w=1	3,7*	3,7*	2	9
立即数与累加器，结果送累加器	0 0 1 1 0 1 0 w | data | data if w=1	3	3		
NOT=把寄存器/存贮器取反	1 1 1 1 0 1 1 w | mod 010 r/m	2,7*	2,7*	2	9

续 表

功能	格式	时钟计数		注释	
		实地址方式	保护虚地址方式	实地址方式	保护虚地址方式
串处理：					
MOVS=传送字节/字	1 0 1 0 0 1 0 w	5	5	2	9
CMPS=比较字节/字	1 0 1 0 0 1 1 w	8	8	2	9
SCAS=扫描字节/字	1 0 1 0 1 1 1 w	7	7	2	9
LODS=把字节/字装入 AL/AX	1 0 1 0 1 1 0 w	5	5	2	9
STOS=存放 AL/AX 中的字节/字	1 0 1 0 1 0 1 w	3	3	2	9
INS=从 DX 转接口中输入字节/字	0 1 1 0 1 1 0 w	5	5	2	9,14
OUTS=从 DX 转接口中输出字节/字	0 1 1 0 1 1 1 w	5	5	2	9,14
串处理(续)：					
重复在 CX 中的计数次					
MOVS=传送串	1 1 1 1 0 0 1 0 | 1 0 1 0 0 1 0 w	5+4n	5+4n	2	9
CMPS=比较串	1 1 1 1 0 0 1 z | 1 0 1 0 0 1 1 w	5+9n	5+9n	2	9
SCAS=扫描串	1 1 1 1 0 0 1 z | 1 0 1 0 1 1 1 w	5+8n	5+8n	2	9
LODS=装入串	1 1 1 1 0 0 1 0 | 1 0 1 0 1 1 0 w	5+4n	5+4n	2	9
STOS=存放串	1 1 1 1 0 0 1 0 | 1 0 1 0 1 0 1 w	4+3n	4+3n	2	9
INS=输入串	1 1 1 1 0 0 1 0 | 0 1 1 0 1 1 0 w	5+4n	5+4n	2	9,14
OUTS=输出串	1 1 1 1 0 0 1 0 | 0 1 1 0 1 1 1 w	5+4n	5+4n	2	9,14
控制转移					

续　表

功　能	格　式	时钟计数		注　释	
		实地址方式	保护虚地址方式	实地址方式	保护虚地址方式
CALL=Call:					
段内直接	11101000 | disp-low | disp-high	7+m	7+m	2	8
寄存器/存贮器段内直接	11111111 | mod 0 1 0 r/m	7+m, 11+m*	7+m, 11+m*	2	8,9
段间直接	10011010 | segment offset	13+m	26+m	2	8, 11, 12
	segment selector				
仅对保护方式(段间直接):					
通过调用门到相同特权层			41+m		8, 11, 12
通过调用门到不同特权层,无参数			82+m		8, 11, 12
通过调用门到不同特权层,X 个参数			86+4x+m		8, 11, 12
通过 TSS			177+m		8, 11, 12
通过任务门			182+m		8, 11, 12
段间间接	11111111 | mod 0 1 1 r/m　(mod≠11)	16+m	29+m*	2	8, 11, 12
仅对保护方式(段间间接):					
通过调用门到相同特权层					
通过调用门到不同特权层,无参数			44+m*		8, 9, 11, 12
通过调用门到不同特权层,X个参数			83+m*		8, 9, 11, 12
通过 TSS			90+4x+m*		8, 9, 11, 12
通过任务门			180+m*		8, 9, 11, 12
JMP=Unconditional jump:			185+m*		8, 9, 11, 12
短段内直接	11101011 | disp-low	7+m	7+m		8
长段内直接	11101001 | disp-low | disp-high	7+m	7+m		8
寄存器/存贮器段内间接	11111111 | mod 100 r/m	7+m, 11+m*	7+m, 11+m*	2	8, 9
段间直接	11101010 | segment offset	11+m	23+m		8, 11, 12
	segment selector				

续 表

功 能	格 式	时钟计数		注 释	
		实地址方式	保护虚地址方式	实地址方式	保护虚地址方式
仅对保护方式(段间直接):					
通过调用门到相同特权层			38+m		8,11,12
通过 TSS			175+m		8,11,12
通过任务门		15+m*	180+m		8,11,12
段间间接	11111111 | mod 101 r/m (mod≠11)		26+m*	2	8,9,11,12
仅对保护方式(段间间接):					
通过调用门到相同特权层			41+m*		8,9,11,12
通过 TSS			178+m*		8,9,11,12
通过任务门			183+m*		8,9,11,12
RET=Return from CALL					
段内	11000011	11+m	11+m	2	8,9
段内并给 SP 加立即数	11000010 | data-low | data-high	11+m	11+m	2	8,9
段间	11001011	15+m	25+m	2	8,9,11,12
段间并给 SP 加立即数	11001010 | data-low | data-high	15+m	55+m	2	8,9,11,12
控制转移(续)					
JE/JZ=相等/为零跳转	01110100 | disp	7+m or 3	7+m or 3		8
JL/JNGE=小于/不大于等于跳转	01111100 | disp	7+m or 3	7+m or 3		8
JLE/JNG=小于等于/不大于跳转	01111110 | disp	7+m or 3	7+m or 3		8
JB/JNAE=低不/不高于等于跳转	01110010 | disp	7+m or 3	7+m or 3		8
JBE/JNA=低于等于/不高于跳转	01110110 | disp	7+m or 3	7+m or 3		8
JP/JPE=校验/校验为偶跳转	01111010 | disp	7+m or 3	7+m or 3		8

续 表

功 能	格 式	时钟计数		注 释	
		实地址方式	保护虚地址方式	实地址方式	保护虚地址方式
JO=溢出跳转	01110000 disp	7+m or 3	7+m or 3		8
JS=有符号跳转	01111000 disp	7+m or 3	7+m or 3		8
JNE/JNZ=不相等/不为零跳转	01110101 disp	7+m or 3	7+m or 3		8
JNL/JGE=不小于/大于等于跳转	01111101 disp	7+m or 3	7+m or 3		8
JNLE/JG=不小于等于/大于跳转	01111111 disp	7+m or 3	7+m or 3		8
JNB/JAE=不低不/高于等于跳转	01110011 disp	7+m or 3	7+m or 3		8
JNBE/JA=不低于等于/高于跳转	01110111 disp	7+m or 3	7+m or 3		8
JNP/JPO=不校验/校验为奇跳转	01111011 disp	7+m or 3	7+m or 3		8
JNO=不溢出跳转	01110001 disp	7+m or 3	7+m or 3		8
JNS=无符号跳转	01111001 disp	7+m or 3	7+m or 3		8
LOOP=循环 CX 次	11100010 disp	8+m or 4	8+m or 4		8
LOOPZ/LOOPE=在为零/相等期间循环	11100001 disp	8+m or 4	8+m or 4		8
LOOPNZ/LOOPNE=在不为零/不相等期间循环	11100000 disp	8+m or 4	8+m or 4		8
JCXZ=CX 不为零跳转	11100011 disp	8+m or 4	8+m or 4		8
ENTER=进入过程	11001000 data-low data-high L				
L=0		11	11	2	9
L=1		15	15	2	9
L>1		16+4(L−1)	16+4(L−1)	2	
LEAVE=离开过程	11001001	5	5	2	

续 表

功 能	格 式	时钟计数		注 释	
		实地址方式	保护虚地址方式	实地址方式	保护虚地址方式
INT=Interrupt:					
指定类型	11001101 type	23+m		2	
类型 3	11001100	23+m		2	
INTO=溢出时中断	11001110	24+m or 3	24+m or 3	2	
仅对保护方式:		(3 if no interrupt)	(3 if no interrupt)		
通过中断或自陷门到相同特权层			40+m		8,11,12
通过中断或自陷门以与不同的特权层相适应			78+m		8,11,12
通过任务门			167+m		8,11,12
IRET=中断返回	11001111	17+m	31+m	2,4	8,9,11,12,15
仅对保护方式:					
到不同的特权层			55+m		8,9,11,12,15
到不同的任务(NT=1)			169−m		8,9,11,12
BOUND=检测超出范围的值	01100010 mod reg r/m	13*	13* (Use INT clock count if exception 5)	2,6	6,8,9,11,12
处理器控制					
CLC=清除进位	11111000	2	2		
CMC=把进位取补	11110101	2	2		
STC=设置进位	11111001	2	2		
CLD=清除方向	11111100	2	2		
STD=设置方向	11111101	2	2		
CLI=清除中断	11111010	3	3		14
STI=设置中断	11111011	2	2		14

续　表

功　能	格　式	时钟计数		注　释	
		实地址方式	保护虚地址方式	实地址方式	保护虚地址方式
HLT=暂停	11110100	2	2		13
WAIT=等待	10011011	3	3		
LOCK=总线封锁前缀	11110000	0	0		14
CTS=清除任务转换标志	00001111 | 00000110	2	2	3	13
ESC=协处理器交权	0011TTT | mod LLL r/m (TTTLLL是给协处理器的操作码)	9-20*	9-20*	5	17
保护控制					
LGDT=装入全局描述子表寄存器	00001111 | 00000001 | mod 010 r/m	11*	11*	2,3	9,13
SGDT=存放全局描述子表寄存器	00001111 | 00000001 | mod 000 r/m	11*	11*	2,3	9
LIDT=装入中断描述子表寄存器	00001111 | 00000001 | mod 011 r/m	12*	12*	2,3	9,13
SIDT=存放中断描述子表寄存器	00001111 | 00000001 | mod 001 r/m	12*	12*	2,3	9
LLDT=从寄存器/存贮器中装入局部描述子表寄存器	00001111 | 00000000 | mod 010 r/m		17,19*	1	9,11,13
SLDT=把局部描述子表寄存器存放到寄存器/存贮器	00001111 | 00000000 | mod 000 r/m		2,3*	1	9
LTR=从寄存器/存贮器中装入任务寄存器	00001111 | 00000000 | mod 011 r/m		17,19*	1	9,11,13
STR=把任务寄存器存放到寄存器/存贮器	00001111 | 00000000 | mod 001 r/m		2,3*	1	9,11,13
LMSW=从寄存器/存贮器中装入机器状态字	00001111 | 00000001 | mod 110 r/m	3,6*	3,6*	2,3	9,13
SMSW=存放机器状态字	00001111 | 00000001 | mod 100 r/m	2,3*	2,3*	2,3	9
LAR=从寄存器/存贮器中装入访问权	00001111 | 00000010 | mod reg r/m		14,16*	1	9,16
LSL=从寄存器/存贮器中装入段限	00001111 | 00000011 | mod reg r/m		14,16*	1	9,16

续　表

功　　能	格　　式	时钟计数		注　　释	
		实地址方式	保护虚地址方式	实地址方式	保护虚地址方式
ARPL=从寄存器/存贮器中调整已请求的特权层	0 1 1 0 0 0 1 1 | mod reg r/m		10*, 11*	2	9
VERR=验证读访问:寄存器/存贮器	0 0 0 0 1 1 1 1 | 0 0 0 0 0 0 0 0 | mod 100 r/m		14, 16*	1	9, 16
VERR=验证写访问:寄存器/存贮器	0 0 0 0 1 1 1 1 | 0 0 0 0 0 0 0 0 | mod 101 r/m		14, 16*	1	9, 16

脚注：存贮器操作数的有效地址(EA)，是按照 mod 和 r/m 字段来计算的。

若 mod=11，r/m 字段就被作为 REG 字段处理。

若 mod=00，则 DISP=0*，disp-low 和 disp-high 不出现。

若 mod=01，则 DISP=disp-low 符号扩展为 16 位，disp-high 不出现。

若 mod=10，则 DISP=disp-high，disp-low。

若 r/m=000，则 EA=(BX)+(SI)+DISP。

若 r/m=001，则 EA=(BX)+(DI)+DISP。

若 r/m=010，则 EA=(BP)+(SI)+DISP。

若 r/m=011，则 EA=(BP)+(DI)+DISP。

若 r/m=100，则 EA=(SI)+DISP。

若 r/m=101，则 EA=(DI)+DISP。

若 r/m=110，则 EA=(BP)+DISP*。

若 r/m=111，则 EA=(BX)+DISP。

DISP 跟在指令的第 2 个字节之后(需要 data 时在 data 之前)。

*若 mod=00 并且 r/m=110，则 EA=disp-high，disp-low。

段超越前缀

001reg110

reg 的编码如下：

reg	段寄存器
00	ES
01	CS
10	SS
11	DS

REG 的编码如下：

16 位(W = 1)		8 位(W = 0)	
000	AX	000	AL
001	CX	001	CL
010	DX	010	DL
011	BX	011	BL
100	SP	100	AH
101	BP	101	CH
110	SI	110	DH
111	DI	111	BH

由 BP 寄存器寻址的所有操作数的物理地址，均使用 SS 段寄存器来计算。串基本操作、目操作数的物理地址，使用 ES 段寄存器来计算，这种用法不能被超越。

思 考 题

15-1 为什么说 80286 和 8086 是向上兼容的？试从寄存器集、指令系统、存贮器组成、寻址方式和中断机构等方面来说明。

15-2 80287 为 80286 扩展了那些寄存器？80287 支持了那些数据类型？

15-3 试描述实地址方式下，80286 构成物理地址的过程，设段的基地址为 04FBH，偏移量为 0058H，它的物理地址是什么？

15-4 80286 保护虚地址方式下的指示器、和实地址方式下的指示器有什么不同？使用这种指示器，构成物理地址的过程是怎样的？

15-5 选择子的结构是怎样的？对值为 04FEH 的选择子，如何理解它的含义？

15-6 描述子的作用是什么？80286 总共有多少不同种类的描述子？

15-7 段描述子区别了其他描述子的标志是什么？代码段描述子和数据段描述子是怎样区分的？它们的 TYPE 字段各应该作怎样的解释？

15-8 特殊系统数据段描述子有那几种？它们之间是用什么字段来相互区分的？它们的作用各是什么？

15-9 门描述子为什么没有基地址和限，而代之以目选择子和目偏移量？门描述子在使用上同其他描述子相比，有什么最显著的不同？

15-10 80286 是用什么硬件支持对四个当前段的访问的？它们与段寄存器的关系如何？

15-11 试对图 15.32 中在段寄存器和段 CACHE 寄存器中的选择子和描述子作出分析。

15-12 试对图 15.38 中的选择子和描述子作出分析。

15-13 80286 总共支持几种描述子表？它们各能包含什么类型的描述子？为了支持这些描述子表，80286 提供

了那些硬件寄存器？这些寄存器的内容是从什么地方装入的？被装入的内容的数据格式是怎样的？

15-14 设发生了类型为 30H 的中断，试描述 CPU 调用中断处理程序的过程，假定中断描述子表寄存器的内容如图 15.37 所示。

15-15 试对 CPL、DPL、RPL、EPL 作出解释，它们各是由什么数据结构的什么字段决定的，它们的作用各是什么？

15-16 在一个任务中，CPL、RPL、DPL 之间要满足什么关系，才能实现对具有 DPL 值的描述子进行访问？例外是什么？

15-17 80286 中控制转移可以分成几类？在实现这些转移时必须满足怎样的特权条件？

15-18 怎样才能实现在同一任务之内的不同特权层之间的控制转移？简述这个控制转移的实现过程。

15-19 80286 是通过那些途径实现保护的？

15-20 什么是"异常"？异常在保护虚地址方式中的地位如何？

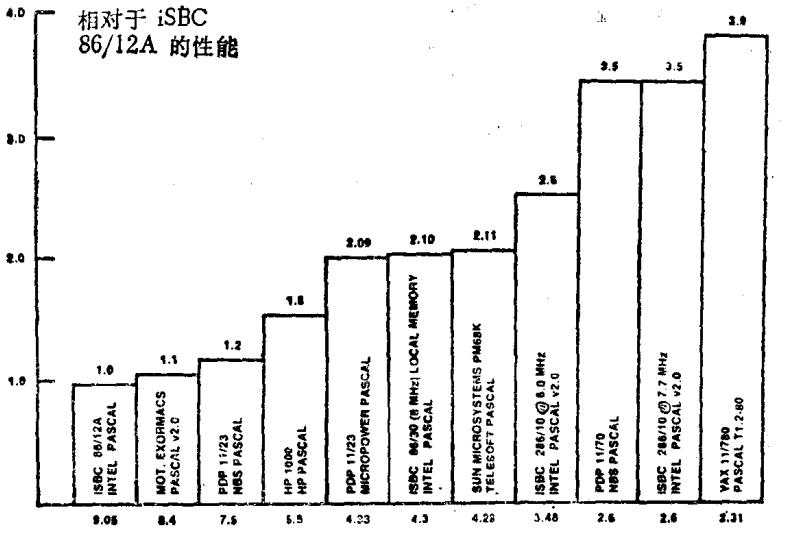
15-21 80286 为任务转换操作，提供了那些硬件和软件机构？任务转换操作的硬件机构所装入的内容，是从何处来的？它们的格式如何？

15-22 任务转换要牵涉到那些操作？任务状态段中的反向链接选择子起什么作用？

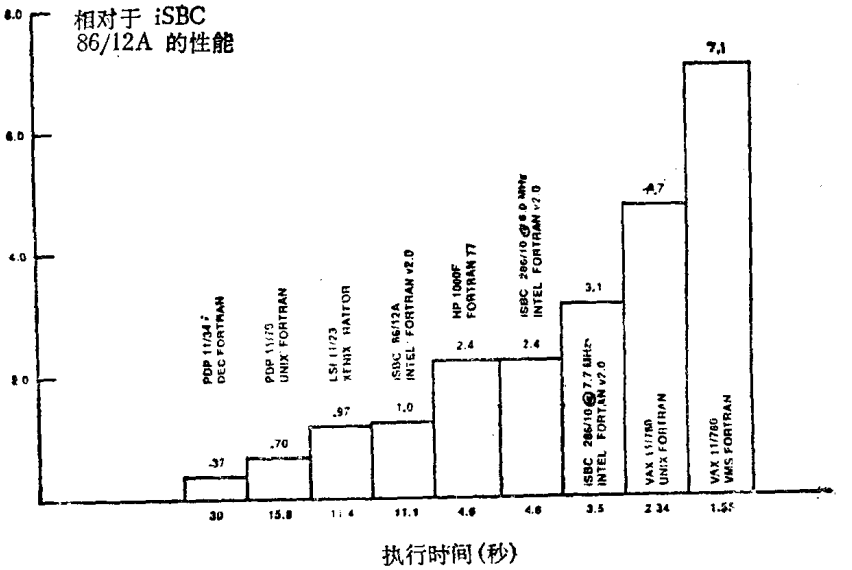
15-23 当一条指令请求访问一个不存在于内存中的段时，80286 要执行什么操作？

第十六章　iSBC 286/10 产品系列

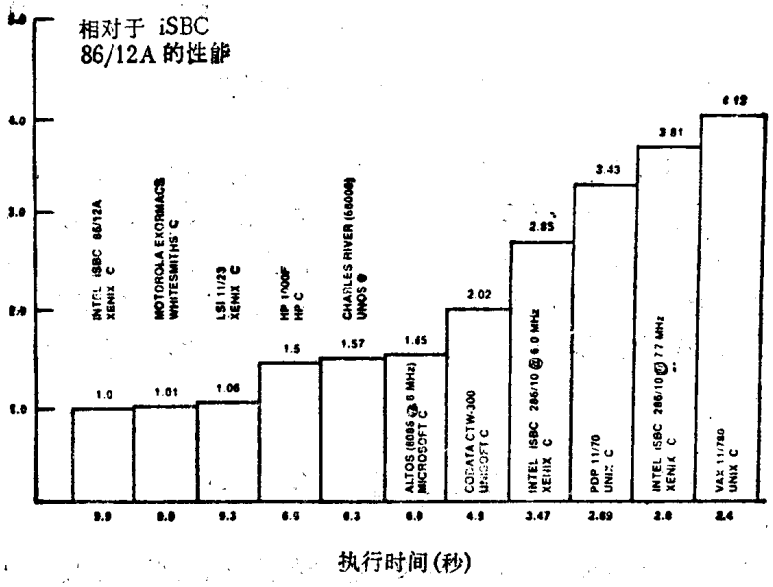
随着微型计算机技术的迅猛发展，其产品又从器件一级上升到板块一级，即构成微机整个系统或系列的主机或系统模块，不再是一般的原器件，而是按照标准预制的各种模板，其主机板实际上就是一台独立的单板机。由上述组合的系统或系列，具有如下特点：组合工艺简便，



(a) 执行 PASCAL 语言的结果



(b) 执行 FORTRAN 语言的结果



(c) 执行C语言的结果

图 16.1 iSBC 286/10 与其他机器的性能比较

将模板插入相应的插座即可；大大提高了功能和功效；可直接使用现成软件；成本大幅度下降，真正做到价廉物美。而最为重要的是，为微机应用的普及，创造了最好的条件。

Intel 公司的 iSBC 286/10 序列，就是这样的一整套主机和系统扩充板。其中 iSBC 286/10 是主机板，它实际上是一台以 80286 为 CPU 的单板机。它的性能优劣与否，在整个序列中起着举足轻重的作用。为了使读者对 iSBC 286/10 单板机的性能，有一个感性认识，我们在图 16.1 中，用直方图来表示该单板机与其它机器，在执行不同语言的 ERATOSTHENES 筛选程序时的性能作一比较。其中(a)，是执行 PASCAL 语言的结果；(b) 是执行 FORTRAN 语言的结果；(c)是执行 C 语言的结果。

§16.1 iSBC 286/10 单板计算机

iSBC 286/10 单板计算机(Single Board Computer)，是 Intel 完整的微型计算机模块和系统中的一种，它采用了 VLSI 技术的优势，从而能把这个完整的微型计算机系统，构造在一块 6.75×12.0 英吋的印制板上。板上包括了 CPU、系统时钟、存贮器插座、I/O 转接口和驱动器、串行通讯接口、优先级中断逻辑以及可程序计时器等。iSBC 286/10 板，是把 iAPX 286 CPU 和 iLBX™ 协总线组合在一起的第一种单板计算机。这种组合，提供了最高性能的 16位微型计算机系统。iLBX 结构扩展，使系统的高性能，在需要大容量系统存器贮的应用中，也得以保持。

§16.1.1 iSBC 286/10 功能概述

iSBC 286/10 板在多总线系统结构中，使用了高效的 iAPX 286 CPU，并由 iLBX 总线来加强。它包括中断、存贮器和 I/O，从而构成了一个完整的单板计算机系统。

§16.1.2 中央处理部件

iSBC 286/10 板的中央处理部件，是以 6.0MHz 时钟频率运行的 80286 CPU。我们知道 80286 CPU 与 iAPX 88 和 iAPX 86 CPU 是向上兼容的，由于 80286 具有并行的片结构，所以它运行起 iAPX 88 和 86 的代码，要比 8088、8086 快得多。另外，80286 提供了在片内的存贮器管理、和寻址达每个任务 1000 兆字节的保护虚拟存贮器。数字处理能力，可用任选的 80287 数值处理器来加强，80286 和 80287 的时钟频率是独立的，80287 的速率可由跳线选择为 4.0 MHz 或 8.0MHz。

§16.1.3 指 令 集

指令集为 80286 的全部指令，包括可变长度指令格式(包括双操作数指令)、8 位和 16 位、带符号和无符号的 2 进制 BCD 和非压缩的 ASCII 数据的算术算符，以及重复字和字节串处理功能。

为了加强数字处理能力，80287 数值数据处理器扩展了 80286 结构和数据集。有 60 多条数值指令提供了算术、三角、超越函数、对数和指数运算。支持的数据类型有 16 位、32 位和 64 位整数、32 位和 64 位浮点数、18 个数字的压缩 BCD 以及 80 位暂存。80287 满足关于数值数据处理的 IEEE P 754 标准，并且保持与基于 8087 的系统相兼容。

§16.1.4 结 构 特 点

iAPX86、88、186 和 286 CPU 系列，都含有相同的基本寄存器集、指令集和寻址方式。80286 处理器与 8086、8088 和 80186 CPU 是向上兼容的。

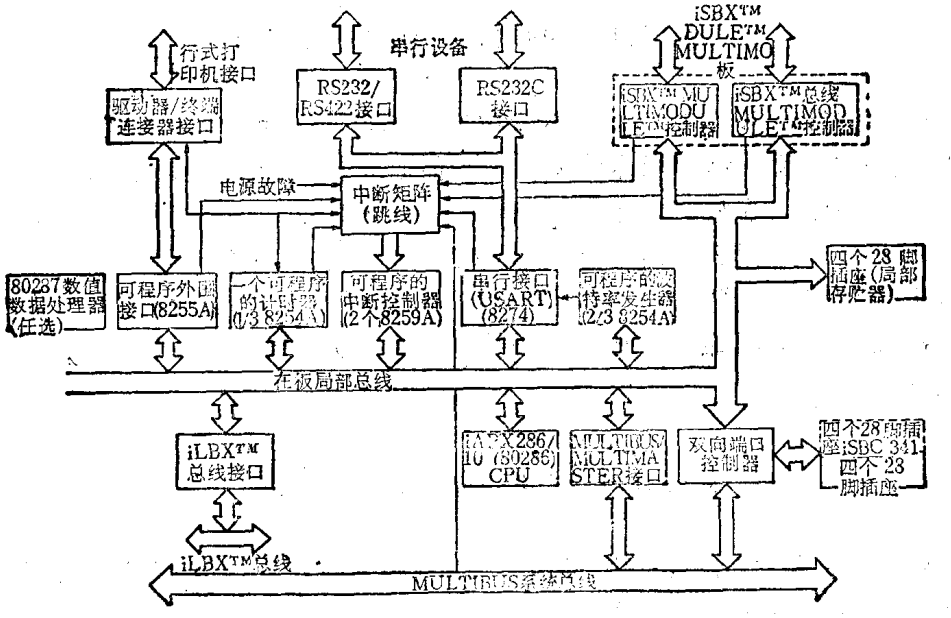


图 16.2 iSBC 286/10 方框图

80286 以两种方式进行操作，即 iAPX 286 实地址方式和保护虚地址方式。在 iAPX 286 实地址方式中，程序可以使用多达 1 兆字节的地址空间。在保护虚地址方式中，程序使用虚拟地址，80286 CPU 能自动地把每个任务的 1000 兆虚拟地址，映射到 16 兆实地址空间。这种方式还提供了存贮器保护，以隔离操作系统，并为每个任务的程序和数据保密。两种方式都提供了相同的基本指令集、寄存器和寻址方式。图 16.2 是 iSBC 286/10 的方框图。

一、向量中断控制

进入 iSBC 286/10 板的中断，由两个装在板上的 8259 A 可程序中断控制器和 80286 的 NMI 引脚处理。最多可以有 16 个中断源，从这些中断源产生的中断，被按优先级排序，然后作为一个向量地址发送给 CPU。通过把在分离的 SBC 板上的 8259 A 从中断控制器串联到在板上的主中断控制器，就可以进一步扩大中断处理的能力。

二、中断源

有 23 种可能的中断源，被连接到中断跳线矩阵，在那里，用户可以把所要的中断源，连接到指定的中断层。表 16.1，表示了由中断机构支持的设备和功能。

表 16.1 中断请求源

设　　备	功　　能	中断个数
MULTIBUS 接口	来自 MULTIBUS 常驻外围设备或其它 CPU 板的请求	8*
8259A 可程序中断控制器	串联至主 8259 A 的 8 层向量中断请求	1
8274 串行控制器	串联至主 8259A 的 8 层向量中断请求	1
8255 A 行式打印机接口	指明输出缓冲器空	1
8254 计时器	计时器 0,1 输出：功能由计时器方式决定	2
iSBX™ 连接器	功能由 iSBX™MULTIMODULE™ 板决定	4 每个 iSBX™ 连接器 2 个
总线故障保安计时器	指出被寻址的 MULTIBUS 常驻设备在 6 微秒内还没有响应命令	1
电源故障中断	指出 AC 电源已不在容许的范围之内	1
外部中断	来自辅助连接器的通用中断通常用作前底板中断	1
在板逻辑	来自边缘触发锁存，反向器，OR 断门的条件中断源	3

* 用在 MULTIBUS 板上的从 8259A PIC 可以扩展到 56 种.

三、存贮器容量

在板上有 8 个 28 脚的 JEDEC 插座，可以用来连接组合的 8 位设备，包括 RAM、iRAM、EPROM 及 E^2PROM。这些插座被分成两个各有 4 个插座的块，其中一个是双向端口块。该块可以连接一个 iSBC 341 JEDEC 插座扩展模块，从而使插座扩展到 8 个（即总共有12个插座）。EPROM 使用 12 块 27128 EPROM，容量是 192 KB，RAM 使用 8K×8RAM，容量是80KB。

四、串行 I/O

iSBC/10 有一个双通道的串行通讯接口，该接口使用 8274 多协议串行控制器 MPSC (Multi-Protocol Serial Controller)。两个独立的用软件选择的波特率发生器，给 MPSC 提供所有公用的通讯频率。协议(异步、IBM 双同步，或 SDLC/HDLC)、数据格式、控制字符格式、奇偶校验和波特率都由程序控制。软件和 MPSC 结合，可以通过询问或中断驱动程序来进行。通道中的一个既能同 RS232C 接口，又能同 RS422 接口相接，而另一个通道则只能与 RS232C 接口相接。每个通道的数据、命令和信号地线都被引向 2 个 26 脚的连接器。

五、可程序计时器

iSBC 286/10 板，提供了 3 个独立的完全可编程的 16 位间隔计时器/事件计数器，它们都使用 8254 可程序间隔计时器。每个计时器都能以 BCD 或 2 进制方式操作。这些计时器/计数器中的两个，可由程序设计者应用于在软件控制下，产生正确的时间间隔。这些计时器的输出方式，是可跳线选择的。该输出可以独立地引向 8259 A 可程序中断控制器，或引向 8274 MPSC，以便对外部事件记数，或提供波特率发生器。第三个在 8254 中的间隔计时器，专门用来为在 iSBC 286/10 板上的 MPSC 串行控制器中的可程序波特率发生器提供一个时钟。系统软件使每个计时器，都能独立地选择所需要的功能。表 16.2，列出了可供使用的七种功能。每个计时器的内容可在系统操作期间的任何时刻读出。

表 16.2 可程序的计时器功能

功　　能	操　　作
终止计数中断	在到达终止计数值时，就发出一个中断请求。这种功能在产生实时时钟方面很有用。
可程序的电平翻转	在获得一个外部触发脉冲边缘，或软件命令的情况下，输出变成低电平，在到达终止计数值时，就返回到高电平状态。这种功能是可重新触发的。
频率发生器	由 N 个计数器来除。在一个输入时钟周期中，输出将变成低电平，并且从一个低电平到下一个脉冲之间所经历的时间是输入时钟周期的 N 倍。
方波频率发生器	输出在到达计数的一半以前，将一直保持高电平，并在另一半计数时间中，一直保持低电平。
软件触发的选通	在软件装入计数(N)之前输出一直保持高电平。在计数装入后，要计数 N 次，输出在一个输入时钟周期期间，变成低电平。
硬件能发选通	在上升沿计数器触发输入之后，输出在一个 N 次计数的时钟周期中，变成低电平。计数器是可重触发的。
事件记数器	在可跳线选择的基础上，时钟输入变成来自外部系统的输入。在计数器"窗口"被允许、或在系统中的 N 个事件发生后可能产生的一个中断之后，CPU 可以读发生的事件数。

六、行式打印机接口

8255 A 可程序外围接口(PPI)，提供一个行式打印机接口。若干种在板功能和四个非专用的输入位，驱动器被提供用来组成一个完整的中心兼容的行式打印机接口。在板功能由 PPI 电源故障传感、超越、NMI 屏蔽、非易失 RAM 允许、清除超时中断、LED0 和 1、清除边缘传感触发、MULTIBUS 中断和串行通道 A 的循环返回等组成。PPI 的 I/O 线被分成三个 8 位转接口，A、B、C。四个非专用输入位，使四个由用户构成的跳线连接的状态能输入。当把转接口 A 和 C 用于输出、而 B 用于输入时，PPI 必须被编程为方式 0。把一个"哑元"写入转接口 B，用来设置 iSBC 286/10 板进入保护方式。并行转接口位的分配，表示在表 16.3 中。

表 16.3 并行转接口位分配

位	功能
	转接口 A——输出
0	行式打印机数据位 0
1	行式打印机数据位 1
2	行式打印机数据位 2
3	行式打印机数据位 3
4	行式打印机数据位 4
5	行式打印机数据位 5
6	行式打印机数据位 6
7	行式打印机数据位 7
	转接口 B——输入
0	通用输入 0
1	通用输入 1
2	通用输入 2
3	通用输入 3
4	行式打印机 ACK/(低电平有效)
5	电源故障传感/(低电平有效)
6	行式打印机错误(高电平有效)
7	行式打印机忙(高电平有效)
	转接口 C——输出
0	行式打印机数据选通(高电平有效)
1	超越/(低电平有效)
2	NMI 屏蔽(0=NMI 允许)
3	非易失 RAM 允许；清除超时中断
4	LED 0(1=开)；清除边缘传感触发
5	MULTIBUS 中断(1=有效)
6	串行 CHA 循环返回(0=工作，1=循环返回)
7	LED 1(1=开)；清除行式打印机 ACK 触发器

§16.1.5 MULTIBUS 系统结构

一、概述

MULTIBUS 系统结构，包括如图 16.3 所示的三总线结构。它们是系统总线、局部协总线和 MUTIMODULE™ 扩展总线。每种总线结构，都能最佳地适应特殊系统的需要。系统总线为包含存贮器、I/O 扩展和支持多处理的通用系统设计提供了基础。局部协总线使 CPU 板能经过一条私用总线进行高效大容量的存贮器访问。MULTIMODULE 扩展总线，是给一个基本 CPU 板增加廉价的 I/O 功能的一种手段。三种总线都安装在 iSBC 286/10 板上，从而提供了一个完整的系统结构。

二、系统总线——IEEE 796

MULTIBUS 系统总线，就是 Intel 工业标准 IEEE 796 微型计算机总线结构。IEEE 796 具有 24 条地址线和 16 条数据线，可以支持 8 位的或 16 位的单板计算机。在它的最简单的应

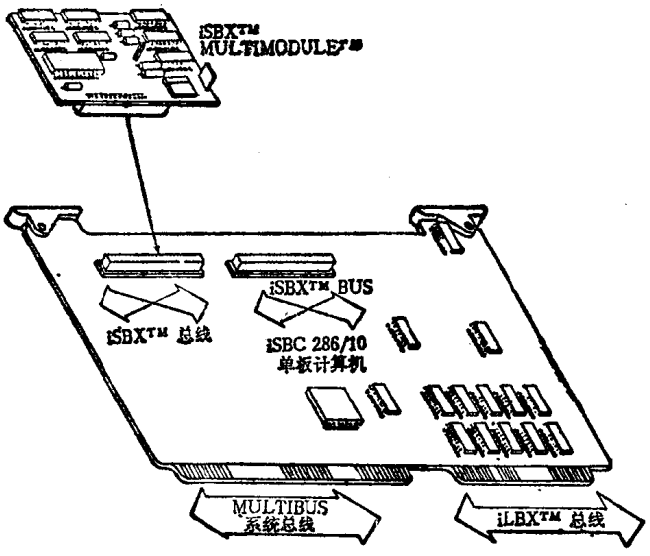


图 16.3 MULTIBUS 系统结构

用中，该系统总线，能使已经包含在单板计算机上的功能(例如存贮器和数字 I/O)得到扩展。当然，IEEE 796 也能有效地配合带有多处理器的分布处理结构，且能从设备 I/O 来解决微机最重要的应用的外围板能力。

三、系统总线扩展能力

使用 Intel MULTIBUS 兼容扩展板，可以扩展存贮器和 I/O 容量，同时能加上附加的功能。存贮器可以加上由用户指定的 RAM 板、EPROM 板或混合板的组合来扩展。使用数字 I/O 和模拟 I/O 扩展板，可以增加输入/输出的容量。加上单密度、或双密度软磁盘控制器，或硬磁盘控制器，可以取得大容量存贮的能力。可供使用的模块、可扩展底板和插件箱，用来支持多板系统。

四、系统总线——多总线主设备能力

对于那些需要附加处理能力和多处理优点(即若干个 CPU 和/或控制器，通过系统总线通讯，逻辑上共享系统任务)的应用，iSBC 286/10 板提供了整个系统总线的裁决控制逻辑。这个控制逻辑能使多至三块 iSBC 286/10 板，或其他总线主设备，包括 iSBC 80 系列的 MULTIBUS 所兼容的 8 位单板计算机，能使用一系列优先级方法来共享系统总线。使用一个外部并行优先级译码器，可以使多至 16 个总线主设备共享该 MULTIBUS 系统总线。使用多总线主设备能力，除了能实现多处理结构方式以外，还为各种形式的 DMA 传送，提供了十分有效的机构。

五、iLBX™ 总线——局部协总线

iSBC 286/10 板，还提供了 MULTIBUS 结构的局部协总线。这种标准扩展，使在板上的存贮器能和不在板上的存贮器一起运行。一块 CPU 板和 iLBX 存贮器板的组合，结构上相等于一个单板计算机，因此可以称为“虚拟 SBC”。iLBX 通过 P2 连接器来实现，并需要用电缆

连接一个系统的虚拟 SBC (见图 16.4)。其他支持 iLBX 总线的 Intel 产品有：

iSBC	028 CX	128 KB	iLBX RAM 板
iSBC	056 CX	256 KB	iLBX RAM 板
iSBC	012 CX	512 KB	iLBX RAM 板
iSBC	428	JEDEC	28 引脚插座板
iSBC	580	MULTICHANNEL™ 接口板	

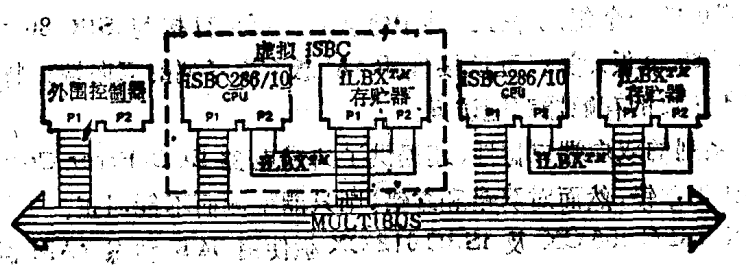


图 16.4 MULTIBUS/iLBX™ 结构

六、iSBX™ 总线 MULTIMODULE™ 在板扩展

在 iSBC 286/10 微型计算机板上，提供了两个 8/16 位 iSBX™ MULTIMODULE 连接器。通过这些连接器，可以增加附加的在板I/O功能。iSBX MULTIMODULE 最佳地支持了由 VLSI 外围器件，例如附加的并行和串行 I/O、模拟 I/O、小容量存贮设备控制器(例如磁带和软磁盘)和其他定制的接口，以满足特殊的需要。由于是直接安装在单板计算机上的，所以在同其他方法，例如用 MULTIBUS 形成兼容板的方法比较时，就具有较少的接口逻辑，较少的功耗，简单的封装，较高的性能和较低的价格等优点。在 iSBC 286/10 板上的 iSBX 连接器，提供了与在板局部总线，包括使数据传送速度达到最大的 16 根数据线，进行接口所必须的全部信号。在 iSBC 286/10 微型计算机板上，还提供了为使用 8 位数据通路和 8 位 iSBX 连接器而设计的 iSBX MULTIMODULE 板。Intel 公司的产品中有可供使用的一个 iSBX MULTIMODULE 任选板系列。

§16.1.6 软件支持

对 iSBC 286/10 板的实时支持，由 iRMX™ 286 R 操作系统提供。iRMX 286 R 是 iRMX 86 核的改编本，以便能在实地址方式的 iSBC 286/10 上操作。iRMX 286 R 加强了支持板结构的 ICU，对在板的 8274 增加了一个驱动程序，并能支持 80287。iRMX 286 R 操作系统与 iRMX 86 完全兼容。

XENIX* 操作系统，将提供交互多用户支持。XENIX 是 UNIX**、System III 的可兼容的版本。

支持 iSBC 286/10 板实地址方式的语言，有 Intel 的 ASM 86、PL/M 86、PASCAL 和 FORTRAN 以及许多第三种 8086 语言。支持虚地址操作方式的语言，有 ASM 286、PL/M 286、PASCAL 和C语言。用这些语言开发的程序，可以经由 iSDM™286 系统调试监督程序，从一

* XENIX 是 Microsoft Inc的商标。

** UNIX 是贝尔实验室的商标。

个 Intel Series III Development System 装入到 iSBC 286/10 板。iSDM 286 监督程序，还提供了对目标程序调试的支持，其中包括断点和存贮器检查。

§16.2 iSBC 028 CX、056 CX 和 012 CX iLBX™RAM 板

iSBC 028 CX、iSBC 056 CX 和 iSBC 012 CX RAM 存贮器板，是 Intel iSBC 存贮器和 I/O 扩展板的完整系列中的一个组成部分。其中每块板都可以直接与 iSBC 80、iSBC 88、iSBC86、iSBC 186 和 iSBC 286 单板计算机直接接口。RAM 板 CX 序列的双向端口特性，使之对 MULTIBUS 和 iLBX™ 接口的存贮器都能访问。

RAM 板 CX 系列的双向端口特性，还提供了纠错电路 ECC(Error Correction Circuity)，该电路能检测并纠正单错，然而对双错和多错，则只能检测而不能纠正。

iSBC 028 CX、iSBC 056 CX 及 iSBC 012 CX 板使用 64 K 动态 RAM 器件，组成 128 K、256 K 或 512 K 字节的读/写存贮器。

由于 iLBX 具有双向端口能力和在板 ECC 特性，所以它们十分适合于用存贮器实现和集中存放精确数字的应用，如财务处理、进程控制及医疗设备等方面的应用。

§16.2.1 功 能 概 述

如在 Intel MULTIBUS 说明中概述的那样，iSBC 028 CX、056 CX 及 012 CX RAM 板在物理和电气上是与 MULTIBUS 接口标准 IEEEP 796 相兼容的。另外，如在 Intel iLBX 说明中概述的那样，RAM 板 CX 系列与 iLBX 总线接口，在物理和电气也相兼容。图 16.5 是用 CX 系列板构成的 iLBX™ 系统。

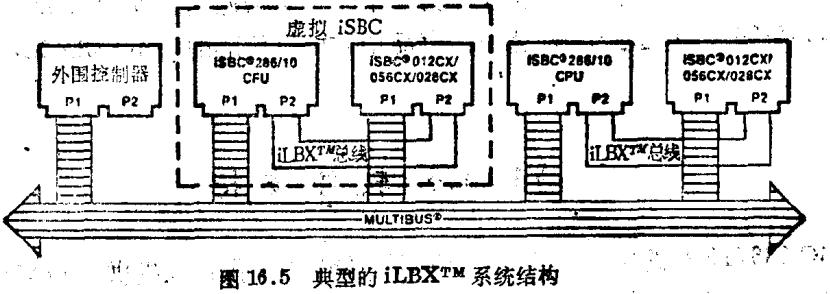


图 16.5 典型的 iLBX™ 系统结构

§16.2.2 双向端口特性

RAM 板的“CX”系列，可由 MULTIBUS 接口或 iLBX 接口访问。Intel 新的局部协总线接口(iLBX 总线)，是非仲裁总线结构，这种结构使 CPU 和存贮器不访问 MULTIBUS 就能相互进行数据的直接传送。iLBX 接口的非仲裁特性，使存贮器访问的时间，得以大大缩短，一般可以比 MULTIBUS 存贮器访问时间快 350% 到 400%。

§16.2.3 系统存贮器容量

使用这个系列板的最大存贮器容量，是 16 兆字节。存贮器的划分，独立于 MULTIBUS和

iLBX 接口。

对 MULTIBUS 进行操作，由在板的跳线给该板指定四个每页为 4 兆字节的页中的一个。每一页又被分成 256 块，其中每块为 16 K 字节，所以在这个系列板中的最小划分，是 16 K 字节。被选中的容量为 4 兆的页中的基地址由跳线指定(最低的 16 K 块)。

iLBX 总线存贮器的划分和 MULTIBUS 的划分的不同之处，在于 iLBX 总线的地址空间，由总线为 16 兆字节的 256 个 64 K 字节的结构块组成。与 MULTIBUS 的划分一样，基地址也是由在板的跳线来指定的。

§16.2.4 检错和纠错

错误检测和纠正，是用 Intel 8206 检错和纠错器件来实现的。ECC 器件与 ECC 检验位 RAM 阵列相连接，以提供检测并纠正一位错和只检双错、大多数多位错的功能。在错误登记由错误状态寄存器 ESR(Error Status Register)支持时，ECC 电路可以通过控制状态寄存器 CSR (Control Status Register)编程，从而改变工作方式。CSR 和 ESR，都通过一个单个的 I/O 转接口，与主 CPU 板通讯。

一、ECC I/O 地址选择

处理器板经由一个单个的 I/O 转接口，与 ECC 电路通讯。这个转接口用于控制状态寄存器 CSR、和错误状态寄存器 ESR。CSR 由用户编程，以便在 ESR 提供存贮器错误信息期间，决定操作的方式。iSBC 028 CX、iSBC 056 CX 及 iSBC 012 CX RAM 板的信息，由一个可程序的阵列逻辑 APL(Programmed Array Logic)器件传送，该器件能选择作为 I/O 转接口的 9 个地址中的一个。真实的选择，是由跳线结构来实现的。在 PAL 中，留有附加的未编程的单元，以便能定义具体的 I/O 地址。

二、控制状态寄存器

在"C"系列 RAM 板中，有 6 种 ECC 的操作方式。每一种方式由 CSR 的软件程序，从主 iSBC 板中获得。这 6 种方式是：

a. 对任何错误都中断方式；
b. 仅对不可纠正的错误中断方式；
c. 纠正方式；
d. 不纠正方式；
e. 诊断方式；
f. 检查伴随字方式。

(1) 对任何错误都中断方式

在这种方式下，当 ECC 电路检测到任何错误(1 位或多位)时，RAM 板将中断 iSBC 处理器板。

(2) 仅对不可纠正的错误中断方式

在这种方式下，仅当 ECC 电路检测到不可纠正的(多位)错误时，RAM 板才中断 iSBC 处理器板。

(3) 纠正方式

在这种方式下,RAM 板将纠正任何可纠正的错误(1 位错)。不能纠正的错误就不进行纠正操作。中断的产生,依赖于所选择的中断方式。

(4) 不纠正方式

在这种方式下,RAM 板不纠正任何错误,ECC 电路连续检测错误,但不采取纠正错误的动作。中断同先前描述的一样。

(5) 诊断方式

这种方式用来测试在板上的 ECC 电路。在这种方式下,选通入 ECC RAM 阵列的写允许被继续禁止。诊断方式可以用来仿真错误,并且能与"检查伴随字方式"一起使用,来检查由 ECC电路产生的检验位。

(6) 检查伴随字方式

这种方式与诊断方式一起,用来测试 ECC 存贮器。在这种方式下,伴随位/检验位被分别

表 16.4 错误状态寄存器格式

位 6	位 5	意义	
0	0	错误所在行	0
0	1		1
1	0		2
1	1		3

位 4	位 3	位 2	位 1	位 0	意义	
0	0	0	0	0	错误所在数据位	0
0	0	0	0	1		1
0	0	0	1	0		2
0	0	0	1	1		3
0	0	1	0	0		4
0	0	1	0	1		5
0	0	1	1	0		6
0	0	1	1	1		7
0	1	0	0	0		8
0	1	0	0	1		9
0	1	0	1	0		10
0	1	0	1	1		11
0	1	1	0	0		12
0	1	1	0	1		13
0	1	1	1	0		14
0	1	1	1	1		15
1	0	0	0	0	错误所在检验位	0
1	0	0	0	1		1
1	0	0	1	0		2
1	0	0	1	1		3
1	0	1	0	0		4
1	0	1	0	1		5
1	1	1	1	0	无错误	
1	1	1	1	1	无可纠正的多位错	

在每个读/写周期同步送入 ESR. ESR 的翻译 PROM 转入一种在检查伴随字方式中的透明方式. 这样,就使由 8206 ECC 器件产生的真实的伴随字,能得到检查.

(7) 错误状态寄存器

这个 8 位寄存器,包含了有关存贮器错误的信息.ESR 反映了最近出现的错误. 表 16.4,表示了状态寄存器的格式. 第 5 和第 6 位表示了错误的行, 而第 0 位到第 4 位指出了(16 位数据字或 6 位 ECC 伴随字中)哪一位错了. 第 7 位总是高电平.

§16.2.5 电池后备电源/存贮器保护

这些板都提供了一条辅助电源总线,以便能把分离的电源,接到读/写存贮器的需要后援的 RAM 阵列中去. 从辅助总线连接器中,可以引出一个低电平有效的与 TTL 兼容的存贮器保护信号. 在该信号发出时,能禁止对 RAM 板的读/写访问. 因此,这个输入信号,在掉电过程中,提供了对 RAM 内容的保护.

图 16.6,表示了 iSBC RAM 板 CX 系列的方框图.

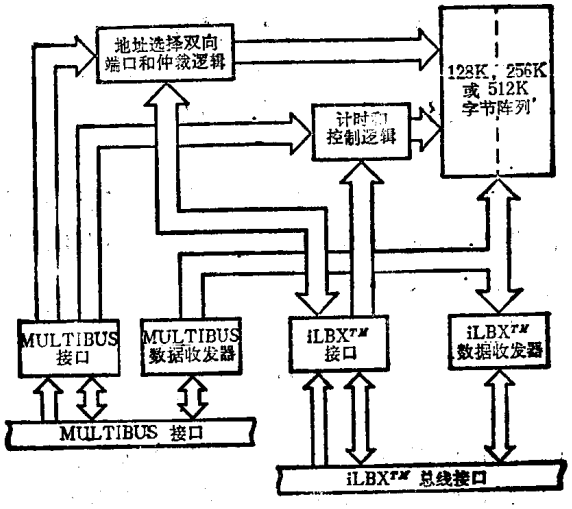


图 16.6 iSBC 028CX/056CX/012CX 方框图

§16.3 iSBC 428 通用插座存贮器扩展板

iSBC 428,是完整的存贮器和 I/O 扩展板系列中的一个组成部分. iSBC 428 通用插座存贮器扩展板,可以通过 MULTIBUS 系统总线,直接与 iSBC 80、iSBC 88、iSBC 86 单板计算机直接接口,用以扩展所需要的存贮器;而 iSBC 286 单板计算机的系统存贮器所需要的扩展,可以经由 MULTIBUS、或高速的 iLBX™ 总线来接口.

§16.3.1 功 能 概 述

iSBC 428 包含 16 个 28 脚插座. 该板的真实容量,由用户插入的器件的类型和数量所决定. iSBC 428 可与 5 种不同类型和密度的器件相兼容: 2K×8 到 64K×8 EPROM/ROM 器

件；2K×8 到 8 K×8 NVRAM (Non-Volatile RAM)器件；2 K×8 到 32 K×8 SRAM 器件；8 K×8 IRAM(Integrated RAM)器件；另外，该板也能由 MULTIBUS 系统总线，或 Intel 的新的 iLBX 总线进行访问。图 16.7，是 iSBC 428 的方框图。

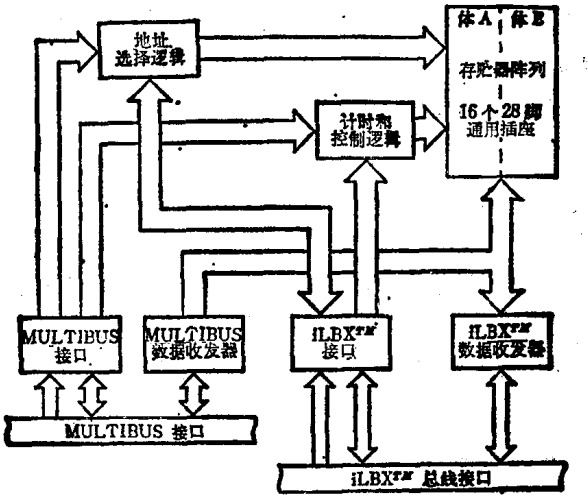


图 16.7 iSBC 428 方框图

§16.3.2 iLBX™ 总 线

iSBC 428 总线，可以通过跳线选择而实现与 MULTIBUS 接口或 iLBX 总线接口通讯。由于 iLBX 总线的专用和非仲裁结构，通过 iLBX 总线接口(与 MULTIBUS 接口相对)进行存贮器访问，其所需时间能大大减少。

§16.3.3 存 贮 体

iSBC 428 的 16 个插座，被分划成各有 8 个插座的两个体。在每个体内，8 个插座又进一步被分划成各有 4 个插座的 2 个组。每组通过一个“配置子”，配置成上述六种器件类型。“配置子”是一种跳线的接通排列，它为具有 4 个插座的每个组进行配置。在每个体内的器件，必须具有相同的密度，而在每个组内的器件，则必须有相同的类型(即 SRAM 或 EPROM)。

§16.3.4 存贮器寻址

iSBC 428 是按页寻址的。共有 64 个 256 K 的页，它们是可跳线选择的。每一个体都可以独立地寻址，并且能驻留在任何页内。真实的起始和结尾地址，是真实器件容量的函数，并且和页本身一样，要由跳线来决定。由于是分页的存贮器寻址结构，所以在一个系统中，可以有多块 iSBC 428 板。

§16.3.5 操 作 方 式

iSBC 428 可以用两种方式中的一种进行操作，这两种方式是：8 位方式、或 8/16 位方式。

8 位方式，为只处理 8 位数据的系统，提供了最有效的存贮器配置；8/16 位方式，使 iSBC 428 板能与使用 8 位和 16 位主机的系统相兼容。操作方式由在板上的跳线选择，并且可以应用于 MULTIBUS 和 iLBX 总线结构。

§16.3.6 存贮器访问

iSBC 428 具有可跳线选择的访问时间，它使该板能与插入到 iSBC 428 板上的特定器件的特性相配合。该板可由跳线配置成能接受访问时间从 50 ns 到 500 ns 的器件，加上一个 50 ns 的形成时间，使在板上的访问时间，为从 225 ns 到 775 ns。

§16.3.7 中　断

iSBC 428 具有为 E^2PROM 的写和擦去操作产生一种中断的能力。中断可以用两种方法配置：一种是通知 E^2PROM 的写周期完成；第二种是在写程序时间里，使系统能查询，以决定 E^2PROM 的状态。

§16.3.8 禁　止

iSBC 428 提供了禁止信号，以便能在启动或诊断操作时，实现 ROM 和 RAM 的重叠。iSBC 428 的每个体，都能由在板上提供的跳线，以系统 RAM 来重叠。

§16.3.9 电池后备电源

iSBC 428 通过一个在板上的连接器，支持电池后备电源操作。分离的电源，通过一条辅助的电源总线，通向需要电池后备电源的系统存贮器阵列。这种辅助电源总线的选择，是由在板上的跳线作出的。

从辅助总线连接器中，可以引出一个低电平有效的与 TTL 兼容的存贮器保护信号，该信号被指定时，能禁止对 RAM 板的读/写访问。这种输入信号，在掉电过程中，可以提供对 RAM 内容的保护。

§16.3.10 支持器件

表 16.5 iSBC 428 支持的器件

类　型	容量 512×8	2K×8	4K×8	8K×8	16K×8	32K×8	64K×8	注　解
EPROM	—	2716	2732 A	2764	27128	27256	×	—
ROM	—	×	×	×	×	×	×	—
EEPROM	—	2817 A	×	×	×	×	—	5V，加强的
SRAM	—	×	×	×	×	×	—	NMOS & CMOS
NVRAM	—	×	×	×	—	—	—	—
IRAM	—	—	—	2186	—	×	—	—

×——表示 iSBC 428 将支持所指出的器件，但该器件目前 Intel 公司还未提供。

表 16.5 中所列出的是，由 iSBC 428 支持的当前和未来的器件。

§16.4 iSBC 580 MULTICHANNEL™ 总线到 iLBX™ 总线的接口

iSBC 580 接口板，是 Intel 的 MULTIBUS 微型计算机完整系列中的一个组成部分，它使用于系统内部通讯(MULTIBUS 系统总线)，高速 I/O(MULTICHANNEL™ DMA I/O 总线)、扩展 I/O(iSBX™ I/O 扩展总线) 以及高速存贮器扩展 (iLBX™ 执行总线) 等分离的优化总线，把系统性能提到了最高的水平。在增强的 MULTIBUS 系统结构中，iSBC 580 通过把一个 MULTICHANNEL I/O 总线到 iLBX 总线的接口，组合在一块 6.75×12.00 英吋的印刷电路板上，从而提供了一个关键的组件。把一个 LSI 状态的机器和标准的在片固件一起使用，可以使吞吐量达到最大。在板上的 Intel 8048 单片微型计算机在 MULTICHANNEL 控制器、设备和常驻 iLBX 总线的多达 16 兆字节的存贮器之间，以高达每秒 5.3 兆字节的速率传送数据。由于 iSBC 580 板起了MULTICHANNEL的发信者/收信者的作用，它通过不使用 MULTIBUS 系统总线，而在 MULTICHANNEL I/O 总线和系统存贮器之间传送数据，因而能提高整个系统的功能。如图 16.8 所示，这样做的结果，就可以使其他系统任务在 I/O 块传输的同时，利用 MULTIBUS 资源。该板的高吞吐量和独立于 MULTIBUS 的活动，使它适用于在要把大量的数据输入 MULTIBUS 系统、或从该系统中输出的应用。例如 MULTIBUS 对主计算机的连接、大容量存放、图象显示以及高速数据拦截子系统的接口等。

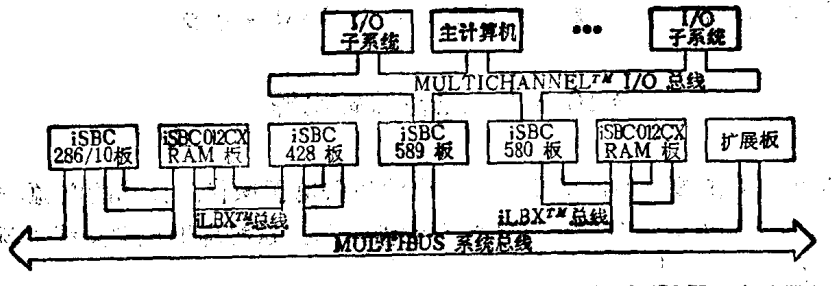


图 16.8 iSBC 580 板，被构成一个 iLBX™ 总线的基本主设备，在 iLBX™ 存贮器和 MULTICHANNEL™ 设备之间传送数据，而不使用系统总线。iSBC 589 板起了对 MULTICHANNEL™ 的监督作用，并在 MULTIBUS 存贮器和 MULTICHANNEL™ 设备之间，进行数据传送。

§16.4.1 MULTICHANNEL™ 接口能力

MULTICHANNEL I/O 总线，是用来在一个微型计算机系统，和多至 15 块传送设备之间，提供一条通用、高速的数据通路的。使用 16 位宽的数据总线、和一种简单的异步信号交换方法，MULTICHANNEL 总线可以在长达 15 米的距离之内，以每秒 8 兆字节的最大字符组吞吐量进行操作。该总线由 16 根地址/数据线、6 根控制线、2 根中断线、校验线和复位线所组成。通过这些信号，可以构成 MULTICHANNEL 的管理程序或控制程序，然后初始化一块与在总线上的任何其他设备进行传递的数据。

iSBC 580 板在 MULTICHANNEL I/O 总线上，仅起了一个 16 位的发信/收信设备的作用。作为发信/收信设备，该板要响应由 MULTICHANNEL 管理程序(典型地为一块 iSBC 589

板)、或由 MULTICHANNEL 控制器发出的寄存器读或写以及 DMA 请求。

iSBC 580 板提供了 32 个 MULTICHANNEL 设备寄存器。前三个寄存器，是标准的 STO 状态、SRQ 状态和 SRQ 屏蔽寄存器。其余的寄存器，用来与在板的固件通讯和存放用户数据。可以由写到命令寄存器来启动的固件操作，列在表 16.6 中。

表 16.6 iSBC 580 固件命令

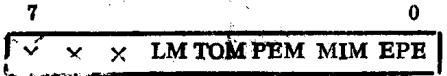
命令代码（十六进制）	操作
0	无操作
1	转为永久脱机
2	STO 查询(诊断)
3	SRQ 查询(诊断)
4	设置在板计时器
5	读在板计时器
6	启动在板计时器
7	停止在板计时器
8	产生任务完成中断
9	在固件上实现检验和(诊断)
A	打开在板 LED
B	关闭在板 LED
C	复位
D, E	保留
F	设置中断屏蔽
10	读中断屏蔽
11-1F	保留

iSBC 580 板，总是从 MULTICHANNEL 接口发送、或接收一个 16 位字，但 iSBC 580 设备寄存器（见表 2）都是 8 位的。寄存器写操作只使用低 8 位(AD_0—AD_7)。寄存器读操作把数据放在 MULTICHANNEL I/O 总线的低 8 位上，而把高 8 位数据线置为 0FFH。

表 16.7 iSBC 580 MULTICHANNEL™ 设备寄存器集

寄存器	地址
STO 状态	00H
SRQ 状态	01H
SRQ 屏蔽	02H
保留	03H-0FH
通用寄存器	10H*-1FH

* 10H 用作命令寄存器



× ：忽略
LM ：iLBX™ 封锁/屏蔽
TOM：定时超出屏蔽(STO)
PEM：奇偶校验错屏蔽(STO)
MIM：MULTIBUS 中断屏蔽 (STO)
FPE：强迫奇偶校验屏蔽

图 16.9 iSBC 580 中断屏蔽寄存器 (14 H)

当 iSBC 580 板，检测到进入 MULTICHANNEL 的数据，有一个奇偶校验错时，或当该板试图寻址不存在的 iLBX 存贮器时，以及该板检测到来自它所在系统的一个 MULTIBUS 中断时，即产生一个可屏蔽的 MULTICHANNEL STO 中断。最后的中断类型，使单板计算机能经由 iSBC 580 板，把一个中断发送到位于另一个 MULTIBUS 系统中的 MULTICHANNEL 管理程序。该板也能在 MULTICHANNEL 总线上，产生一些如图 16.9 所示的 SRQ 中断。

§16.4.2 iLBX™ 总线接口能力

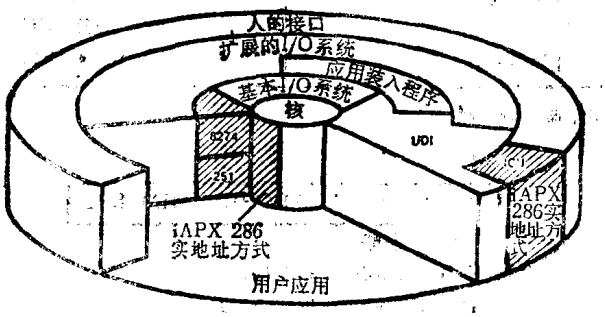
与 MULTIBUS 接口配合使用，iLBX 总线可以用来为单板计算机，提供脱板的存贮器和 I/O 扩展，同时，又能保留在板的性能。iLBX 总线通过准许与基本主机相联的总线的特权使用，能提供对兼容扩展板的高速访问。在最多只有一个且仅是偶尔或非并发地访问 iLBX 资源的辅助主机的情况下，该总线才提供对 iLBX 总线扩展板的有限制的访问。iLBX 总线具有 16 根数据线、24 根地址线、加上控制、奇偶校验和中断信号，使用除了专用于 MULTIBUS 接口高地址线的 4 个引脚以外的 P2 连接器的所有引脚。非多用的地址和数据线，以与一个单板计算机的在板资源可比的速度，提供对在最多可有 4 块分离的扩展板上的 iLBX 总线的 16 兆常驻存贮器的访问。

iSBC 580 板在 iLBX 总线上，可以配置成基本主设备或辅助主设备。图 16.8，表示了一个典型的系统配置，其中一块 iSBC 580 板，起了基本主设备的作用。该板可以访问多达 16 兆字节的 iLBX 存贮器。为了支持在 MULTICHANNEL 总线上的 16 位传送，该板在偶字节 iLBX 地址界上，作 16 位字存贮器访问。为了提高 iLBX 存贮器读操作的性能，iSBC 580 板在当前数据字正在 MULTICHANNEL I/O 总线上传送时，就能从存贮器中预取数据。

§16.5 iRMX™ 286 R 操作系统

§16.5.1 概　　述

高性能的 iRMX™ 286R 操作系统的功能，与 iRMX 86 Relcas 5 操作系统软件结合在一起，可以用来管理和扩展 iSBC 286 单板计算机的资源及其他以 iAPX 286 实地址方式为基础



在iRMX™ 286R中新出现的

图 16.10　iRMX™286 R 层次结构

的微型计算机。iRMX 286 R 操作系统，是一种容易使用、实时、多任务和多程序软件系统，它具有执行在 iAPX 286 实地址方式下，和 iSBC 286/10 单板计算机中的 iRMX86 软件的所有结构层的能力。

这个系统新增加的特点，包括对 8274 的二个通道都给予驱动程序，以及给 iSBX™251 MULTIMODLE™ 板的驱动程序。图 16.10 表示了这种情况。其中阴影部分，是 iRMX™286 相对于 iRMX 86 新增加的部分。

iRMX™286R 操作系统，是一集完整的系统软件模块，它提供了大多数计算机系统需要的资源管理功能。在这里，我们着重描述由 iRMX 286 操作系统所提供的新的特点。这些新的特点加上专门为 OEM 设计的新的能力，就能取得最新的 VLSI 微处理器 iAPX 286 在速度和性能上的好处。

iRMX 286 R 操作系统的新的驱动程序包括：

——一个 8274 终端设备驱动程序，以支持对在 RS 232 方式下，iSBC 286/10 板的双通道异步操作、实行支持的 8274。

——对作为定制的驱动程序安装的 iSBX™251 磁泡存贮器系统的支持。

8274 终端设备驱动程序，支持两个在 RS232 方式下使用的串行通道，并支持所有由 iRMX 86 终端支持的特性。使用 RMX 调试程序，则需要一个 iSBX 351 MULTIMODULE™。

iRMX 286 R 操作系统，是 iRMX 86 操作系统的一种自然扩展，通过简单的升级，给用户提供了更高的性能。操作系统中的实用交互结构，已被加强，以便使以 iAPX 286 微处理器为基础的计算机，例如 iSBC 286/10 单板计算机配置起来更为容易。

§16.5.2 功 能 描 述

为使计算机在同一时间执行多功能的应用中，取得 iAPX 286 微处理器的最佳效益，iRMX 286 R操作系统，提供了一个多程序环境，在其中可以运行许多独立的多任务应用程序。每种应用环境，可以分开进行处理，这样，应用程序员在独立地为每个应用开发或测试软件时，具有能分开管理每个应用程序的资源的灵活性。

iRMX 286 R 系统的资源管理功能，由若干可配置的软件层支持。由最内层，即核支持的许多种功能，是所有的系统都需要的，其他的功能则是任选的。例如，I/O 系统，不必包含在没有辅助存贮器要求的系统中。

iRMX 286 R 操作系统的器件，还提供了系统资源的直接和间接的管理。这些资源常常包括处理器的使用时间和寄存器，80287 数值数据处理器，多达 1 兆字节的系统存贮器，多达 57 种独立的中断源，所有输入输出设备，以及包含在大容量存贮设备中的目录和数据文件。该操作系统提供了多终端和多用户支持软件、改进的 I/O 性能、一个实用的交互配置和一个成熟的事故分析程序，从而使系统设计者，能很容易地设计一个多终端系统。

§16.6 iSDM™286 iAPX 286 系统调试监督包

Intel iSDM™ 286 系统调试监督包，包含了必需的硬件、软件、电缆、PROM 和通过一个高速的连接，把一块 iSBC 286 板、或 iAPX 286 器件应用，与一个 Intellec Series III 接口所需要

的资料。系统调试监督程序，支持一个 iRMX™ 286 实时多任务操作系统，或定制的操作系统的一个 OEM 的选择，用调试工具来检查 CPU 寄存器、存贮器内容、CPU 描述子表和其他关键的环境细节。监督程序能通过内部的 UDI，支持和串行通讯连接，而使程序能访问开发系统的文件。

§16.6.1 功能概述

iSDM 286 系统调试监督包，给以 iAPX 286 为基础的应用的程序员，以从单用户系统到复杂操作系统的新的应用范围所需要的调试工具。程序员可以通过单终端接口、或通过 Intellec Series III 开发系统，访问 CPU 的实地址和保护虚地址。

§16.6.2 通用开发接口

任何 iRMX 86、Series III，或其他以 UDI 为基础的应用，可以由 iSDM 286 监督程序包支持。该监督程序能仿真许多 UDI 调用（实地址和保护虚地址），并把一个文件系统的所有请求，传向主开发站。来自独立的中介软件，例如编译程序和其他可用程序的 UDI 应用，能够立即测试目标 iAPX 286 环境。

§16.6.3 有效的调试命令

有效的命令集使用户可以执行如下的操作：
检查和修改 CPU 寄存器；
检查、修改和传送存贮器单元；
对变量名的符号引用；
寻找和比较存贮器的内容；
设置程序断点；
启动装入应用软件；
单步 CPU 操作；
实地址方式和虚地址方式之间的转换。

§16.6.4 格式化显示

iSDM 286 监督程序能够把所有 iAPX 286 预先定义的数据结构，以清晰的格式进行显示。这种显示给程序员以一种格式化的 CPU 寄存器，如 LDT、GDT、IDT、段选择子和任务状态段的视图，而不仅是一系列没有联系的数字。

§16.6.5 对数值数据处理器的支持

除了能以全部的 NPX 特性，执行 80287 数值协处理器(NPX)的应用外，程序员还可以用 10 进制数和实数格式，检查和修改 NPX 寄存器。存贮器中任何已知含有使用标准实数格式

(IEEEP 754) 数值的单元，可以使用普通的 10 进制记号进行检查或修改。使用这种方法，程序员就可以不必对复杂的实数、整数以及 BCD 16 进制格式，进行编码和译码了。

§16.6.6 高速串行连接

目标应用硬件，通过一根 19.2K 波特的串行连接电缆，连接到开发系统。所有控制操作和 UDI 文件处理，通过由电缆支持的连接进行。图 16.11 表示该串行连接由开发系统的 iSBC 86 的 USART 转接口支持。

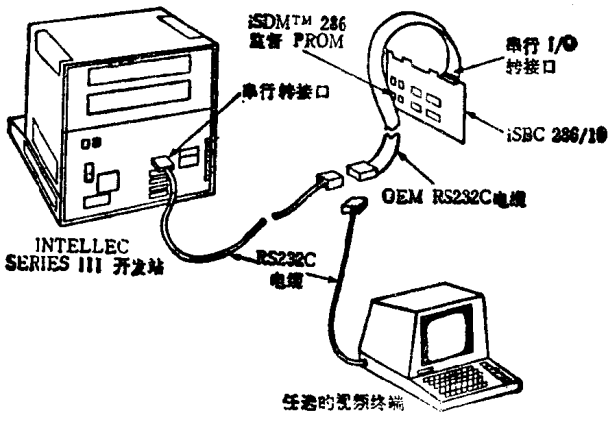


图 16.11 典型的 iSDM™ 286 环境