微计算机丛书

〔日〕中村和夫 井出裕巳 著
陆玉库 译

INTEL 8086 微处理器应用入门

電子工業出版社

责任编辑　王惠民
封面设计　梅　生

科技新[redacted]目56—120
统一书号 15290·3
定　价　1.60元

INTEL 8086

微处理器应用入门

〔日〕中村和夫 井出裕巳 著
陆玉库 译

電子工業出版社

内 容 简 介

本书主要译自〔日〕《インターフェース》杂志，1982年第8期所刊载的INTEL 8086特集；部分内容选译自〔日〕オーム社出版的《8086の使い方》。

全书包括：8086的结构、指令系统、外围芯片、硬件设计、汇编语言程序设计、多总线及开发系统等内容。书后附录有INTEL 8086的指令详解。

本书内容充实、文字简明扼要、图表丰富。对于已了解8位微计算机的读者来说，该书是学习8086微计算机的一种入门读物；并可作为8086指令参考手册。

本书可供从事16位微计算机设计和使用的科技人员及大专院校有关专业师生阅读和参考。

INTEL 8086
微处理器应用入门
〔日〕中村和夫　井出裕巳著
陆　玉　库　译
※
电子工业出版社出版
中国科技情报所印刷厂印刷
新华书店北京发行所发行
各地新华书店经售
※
1983年6月第一版　　开本：850×1168 32开
1984年9月第二次印刷　印张：11 1/8
印数：25,000—73,500册　字数：294千字
统一书号：15290·3　　本社书号：P31·03
定价：1.60元

译 者 的 话

自从美国INTEL 公司于1971年制造出了单片的 4 位微处理器4004以来， 微处理器已经历了十多年的发展历程。 在这十多年的过程中，微处理器不仅在各个领域里得到了广泛的应用，而且微处理器本身不断更新换代，表现了它的强大生命力。继 4 位、 8 位微处理器之后，INTEL公司于1978年又首先发表了 16 位 微 处 理 器8086，此后ZILOG公司、MOTOROLA 公司又相继发表了16位微处理器Z8000 (1979)、MC68000(1980)，使微处理器应用进 入 了一个新阶段。由于16位微处理器不仅具备以前微处理器所有优点，还吸取了不少小型计算机结构设计上的特点，从而使它的处理能力进一步加强，使微型机和小型机差别日趋减小，并有取代小型机的趋势。

目前，我国在广泛应用 4 位和 8 位微计算机的基础上已开始重视16位微机的研制开发和应用。但是在实际工作中深感资料不足。译者根据当前国内微机发展形势的需要，编译了《Intel8086微处理器应用入门》一书。如果本书能对于从事微机研制和应用的技术人员有所启发和帮助，为我国微机的发展起到点滴作用的话，将是译者的最大荣幸。

本书 1～6 章译自日文杂志《インターフェース》1982 年第 8 期； 7、 8、 9 章选译自〔日〕井出裕巳著《8086の使い方》一书。

北京工业大学计算中心李礼贤副教授对译文作了校订，并提供了附录资料，在此谨表谢意。

由于译者水平有限，缺乏实践经验，不妥之处，谨請读者批评指正。

1983.4.

前　言

本书是以打算采用8086微处理器的人为对象，以8086的结构、指令系统、硬件设计、汇编语言程序设计及多路总线等在系统设计中应注意的事项为主要内容，并加以详细的说明。

书中所列举的CP/M操作系统，多总线为 DIGITAL RESEACH公司和INTEL公司的专利。

中村和夫

目　录

1 8086 微处理器的结构

8086是最早出现的第三代16位微处理器，它采用了8位微处理器中所没有的许多新结构。为此，本章首先对8086的结构进行概要的说明。

1.1 8086结构概要

1.1.1 地址总线和数据总线

8086是16位微处理器，其数据总线为16位，地址总线为20位。因此，存储器可以寻址1Mb（1兆字节)的地址空间。在进行I/O（输入/输出）存取时，20位的地址线中只有低16位有效。这样，I/O地址空间为64Kb（64千字节)。图1.1为8085A与8086的数据总线和地址总线的比较，从中可以看出8086的扩充情况如下：

总　线	8085A	8086
数据总线	8位	16位
地址总线（用于存储器）	16位	20位
地址总线（用于I/O）	8位	16位

1.1.2 以字节或字为单位的数据处理

在8086的所有运算指令和传送指令中，可以按字节为单位，也可以按字为单位处理数据。例如，16位数据传送指令可以传送8位数据；带符号或不带符号的乘除运算，可按8位进行，也可以按16位进行。为此，在这些运算指令和传送指令的OP(操作）码中设置

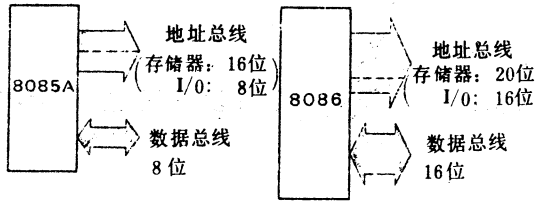


图1.1 8086与8085A的地址总线和数据总线

有W字段（段长为1位），当令W＝0时为字节处理，而令W＝1时为字处理。W字段在操作码中的位置如图1.2所示。在BCD（二-十进制）码数据和ASCII码（美国信息交换标准码）数据的运算时，只能按8位进行，这一点应该注意。

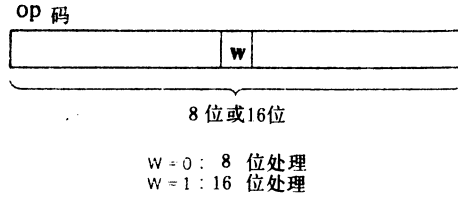


图1.2 W字段

8086中对存储器和I/O进行存取的情况可用图1.3来表示。图中（a）、（b）为字节处理的情况，其中（a）表示存取低位字节（D_7～D_0），（b）表示存取高位字节（D_{15}～D_8）。（c）为字（D_{15}～D_0）处理的情况。（d）也是一种字处理的情况，但它所存取的字跨存储器的两个地址，即包括一个地址的高8位和下一个地址的低8位的内容。8086对这样的字以一条指令进行处理时，要分两次对存储器进行存取，将两个地址分别所对应的高8位和低8位数据读出或写入。

在8086芯片单独使用时，除了乘法指令的积和除法指令的被除

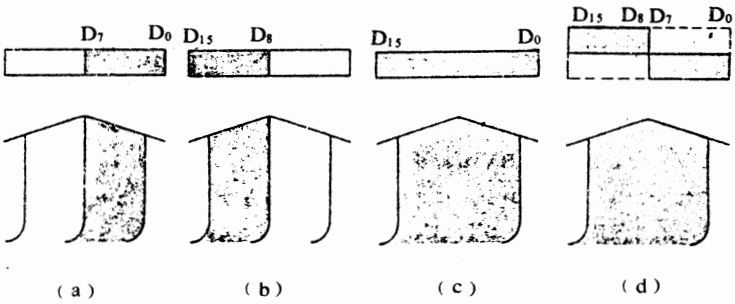


图1.3 8086数据处理的分类

数之外，都不能采用32位数据。如果采用数值运算处理器8087，不仅可以进行32位数据运算，也可以进行80位浮点运算。

1.1.3 以字节为单位的地址分配和$\overline{BHE}$信号

8086对存储器或I/O存取时，有图1.3所示的四种情况，其中(a)、(b)、(d)只能对存储器或I/O存取8位数据。因此，在写入时，必须注意不得改变其他8位数据的内容。为了解决这个问题，8086以字节为单位分配存储器和I/O的地址，并采用$\overline{BHE}$总线高8位允许信号以实现对高位和低位字节的选择。

图1.4表示存储器及I/O的地址分配情况，图中16位数据的低8位（D_7～D_0）分配在偶数地址，而高8位（D_{15}～D_8）分配在奇数地址。这样，通过20位的地址总线能够存取的存储器空间（00000_{16}～$FFFFF_{16}$）为1 Mb或512千字（字长16位）。低8位的偶数地址与8085A具有兼容性。比如在8085A上执行下述程序时，HL寄存器中装入的内容为：$H=12_{16}$，$L=34_{16}$。

```
        ORG 1234H
ADR EQU $
        ⋮
        LXI H,ADR
```

Z 8000、68000刚好与此相反，高8位为偶数地址。

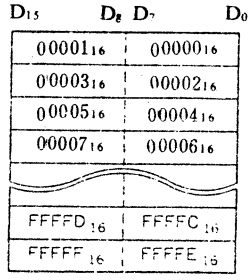


图1.4 以字节为单位的地址分配

表1.1表示8086微处理器执行存储器或I/O存取时与 A_0、$\overline{BHE}$ 信号间的关系。$\overline{BHE}$ 为表示存取 D_{15}～D_8 的信号，$\overline{BHE}$ 从8086输出为低电平有效。正如图1.5的逻辑图所示，通过地址译码器译码的片选信号与 A_0 及 $\overline{BHE}$ 信号相“与”作为相应存储器或I/O的片选信号，就可分别存取高8位（D_{15}～D_8）或低8位（D_7～D_0）数据。

表1.1 8086的存储器或I/O存取形式

A_0	$\overline{BHE}$	存取形式
0	0	存取16位（D_{15}～D_0）数据
0	1	存取低8位（D_7～D_0）数据
1	0	存取高8位（D_{15}～D_8）数据
1	1	不使用

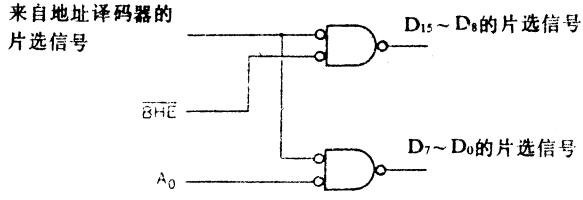


图1.5 片选信号逻辑图

1.1.4 8个16位通用寄存器

8086微处理器有8个16位通用寄存器，参阅图1.6(b)。其中AX～DX寄存器都分成高8位和低8位两部分，它们也可以分别作为独立的8位AL～DH寄存器来指定。图1.6（b）和（a）给出了8086与8085A寄存器的比较，（其中阴影部分与8085A相同）对此需要说明下述几点。

图1.6 8085A与8086内部寄存器的组成

（1）AL～DH寄存器（AH除外）对应于8085A的A～D寄存器。在8085A中只有A寄存器能作为累加器使用，而8086中AL～DH寄存器（8位）和AX～DI寄存器(16位)都可以作累加器使用。

（2）8086的BX寄存器和8085A的HL寄存器相对应。(在8086中，作为寄存器间接寻址使用的地址寄存器有BX、SI、DI寄存器。)

（3）AL～DH寄存器用于8位数据存取时，它们不再是高位字节、低位字节的关系。（在有些微处理器中，不能进行高位字节寄存器和低位字节寄存器间的传送和运算。）

（4）与8085A使用的程序计数器（PC）相对应，8086采用了名称不同的指令指针（IP），因此用监控程序调试时要注意。

上面已经说明AX～DI寄存器和AL～DH寄存器都可以作累加器使用，即可以用下述指令进行操作。

```
ADD    CX,   DX      ; CX←—CX+DX
SUB    SP,   2       ; SP←—SP-2
AND    DH,   AL      ; DH←—DH·AL
```

这些通用寄存器在有些指令中有时还起一些特殊作用。表1.2归纳了这些寄存器的作用。根据它们的特点分别规定了不同的名称，这些名称均采用了该寄存器的英文字头。例如：

AX：（Accumulator）累加器

BX：（Base Register）基址寄存器

CX：（Count Register）计数寄存器

DX：（Data Register）数据寄存器

从表1.2的说明中就可以理解这些寄存器的意义和使用方法。

表1.2 通用寄存器的特殊用法

AX，AL	· 在乘除指令中作累加器 · 在输入输出指令时，作为数据寄存器
AH	· 在LAHF指令（(AH)←(标志)）中作目的寄存器
AL	· 在BCD，ASCII码数据运算时作累加器 · 在XLAT指令（(AL)←((AL)+(BX))中作累加器
BX	· 在间接寻址时作地址寄存器 · 在间接寻址时作基址寄存器 · 在XLAT指令（(AL)←((AL)+(BX)）中作基址寄存器

续表1.2

CX	· 在循环、字符串指令中作循环次数的计数寄存器（指令执行后，CX 寄存器内容发生变化）
CL	· 作为移位、循环指令的移位数、循环数的计数寄存器（指令执行后，CL 寄存器的内容不变化）
DX	· 输入输出指令间接寻址时，作为地址寄存器 · 在乘除指令中作辅助累加器（当乘法的积为32位以及除法的被除数为32位时，存放高16位）
BP	· 在间接寻址时作为基址寄存器
SP	· 作堆栈指针
SI	· 在间接寻址时，作为地址寄存器 · 在间接寻址时，作变址寄存器 · 在字符串处理指令中，作源变址寄存器
DI	· 在间接寻址时，作地址寄存器 · 在间接寻址时，作变址寄存器 · 在字符串处理指令中，作目的变址寄存器

（注）此外，AX、AL寄存器作为累加器，在与立即数进行运算时，比用其他寄存器时指令短一个字节。

图1.6（b）中在指令指针（IP）的下面有4个16位寄存器，这些寄存器称为段寄存器。从图中可以看出这些寄存器比其他寄存器画得略向左错开一些，其作用将在本章的1.2中说明。

1.1.5 标志

8086比8085A新增加了3个标志位。在8085A中，标志位是与A寄存器内容一起进栈、出栈的。但在8086中，标志位的进栈、出栈是通过POPF、PUSHF指令进行的。而在中断时，它与指令指针、CS寄存器一起保留在堆栈中。

（1）进位标志CF

运算指令执行之后，当在最高位（字节运算时为D_7，字运算时

为D_{15}）上产生进位、借位时，该标志置1。

（2）全零标志ZF

运算指令执行后，结果为全零时该标志置1。

（3）符号标志SF

运算指令执行后，最高位的值保持原值不变。

（4）奇偶校验标志PF

运算指令执行后，结果数值的低8位中含“1”的位数为偶数时，该标志置1。

（5）辅助进位标志AF

运算指令执行后，低4位上产生借位或者大于10产生进位时，该标志置1。

（6）溢出标志OF

运算指令执行后，结果的数值超过能用2的补码（补数）所表示的范围（字节运算时为－128～＋127，字运算时为－32768～＋32767）时，该标志置1。

（7）方向标志DF

该标志用于指定字符串处理指令的方向。当DF＝0时，字符串处理由低位地址向高位地址处理；而当D＝1时，则从高位地址向低位地址处理。该标志通过STD、CLD指令来置位、清除。

（8）中断允许标志IF

该标志用于控制硬件中断（不可屏蔽的中断除外）。在IF＝1时可以接受中断；IF＝0时中断被屏蔽，不能接受中断。该标志通过STI和CLI指令置位和复位。

（9）陷阱标志TF

该标志用于控制单步中断。在TF＝1时，如果执行指令就产生单步中断。

8086复位后，段寄存器、标志等的内容将被置位或复位。即：

标志位	全部清除
IP寄存器	0000_{16}

CS寄存器	$FFFF_{16}$
DS寄存器	0000_{16}
SS寄存器	0000_{16}
ES寄存器	0000_{16}
指令队列	清除

程序将从$FFFF0_{16}$开始执行（参阅1.2）。这时通用寄存器的内容是不固定的。

1.1.6 8086的外围芯片

8086芯片上没有时钟发生器、中断控制器等功能。在实际使用8086时，必须外加具有这些功能的芯片。因此，除8086芯片外还备有下述外围芯片。

（1）时钟发生器8284A

该芯片将晶体振荡器产生的信号3分频后，生成L（低电平），H（高电平）=2:1（即占空比为1/3）的时钟信号供给8086。此外，在该芯片中还包括复位脉冲产生电路及使就绪信号与时钟信号同步的触发器。

（2）总线控制器8288

在最大方式（参考1.1.7）时，除了需要存储器、I/O的读出/写入信号之外，还需要产生控制总线的时序信号。由于在最大方式中，必须从8086输出多处理器系统所需要的信号，所以这些信号要通过另外的总线控制器芯片8288输出。由8288输出的信号在时序和电特性上都是与多总线兼容的。在8086的最小方式中，由于只是单一处理器系统，因此不需要8288芯片。

（3）总线仲裁器8289

总线仲裁器8289，也是在8086的最大方式时采用的一种芯片。在多总线中它用于裁决8086与其他模板的处理器、DMA（直接存储器存取）控制器之间的总线使用权。8289与8288一起使用，并且与8288同样输出在时序和电特性上与多总线兼容的信号。

（4）中断控制器8259A

这是一种进行中断控制的芯片。在8085A系统中也使用这种芯片。如果设定成8086方式，就可以用于8086系统。在一块8259A芯片上有8级中断請求输入。通过级联连接还可以扩充到64级输入。

（5）8位锁存器8282/8283

这是一种通用的三态输出8位锁存器。8282为不反相型的，8283为反相型的，I_{OL}最小为32mA（$V_{OL}=0.5V$）。在通常情况下也可以使用74LS373芯片（不反相型的），这时I_{OL}最小为24mA（$V_{OL}=0.5V$）。该芯片引脚连接与8282/8283不同，使用时必须注意。

（6）总线收发器8286/8287

这两种芯片是通用的8位总线收发器。8286为不反相型的，8287为反相型的芯片，I_{OL}最小为32mA（$V_{OL}=0.5V$）。通常也可以使用引脚连接不同的74LS645-1（不反相型）和74LS640-1(反相型)的芯片，I_{OL}最小为48mA（$V_{OL}=0.5V$）。

由于8086中的总线与8085A以同样的形式输出，所以除了上述的外围芯片之外，还采用了8位微处理器的相应芯片。如8251A、8253、8255A、8279、8041A等芯片均可以原封用于8086。

1.1.7 系统的最小和最大方式

根据使用目的不同，8086系统可以分成最小组成方式和最大组成方式，即简称最小方式和最大方式。两种方式的选择不是由程序进行控制的，而是最初由硬件设定的。如将8086 MIN/$\overline{\text{MAX}}$端连接在V_{cc}端上时就是最小方式，连接在地（GND）端上时就是最大方式。

表1.3列出了两种方式的特点。简单地说，最小方式与8085A系统的硬件组成相同，只是将8085A改成了8086。而最大方式是8086才有的，它是以多处理器系统为前提的方式。

图1.7为最小方式连接的举例。图1.8为8085A系统连接的举例。对二者比较可以看出，除了地址总线、数据总线的多少不同之外，几乎输出同样的控制信号。不过二者输出的时钟不同。

表1.3 最小方式和最大方式的特点

最 小 方 式	最 大 方 式
①MIN/$\overline{MAX}$=H电平 ②适于较小规模的单一处理器系统 ③不用总线控制器8288 ④输出与8085A相似的命令信号	①MIN/$\overline{MAX}$=L电平 ②以多处理器为前提的大规模系统 ③需要总线控制器8288 ④输出能直接连接到多总线的命令信号

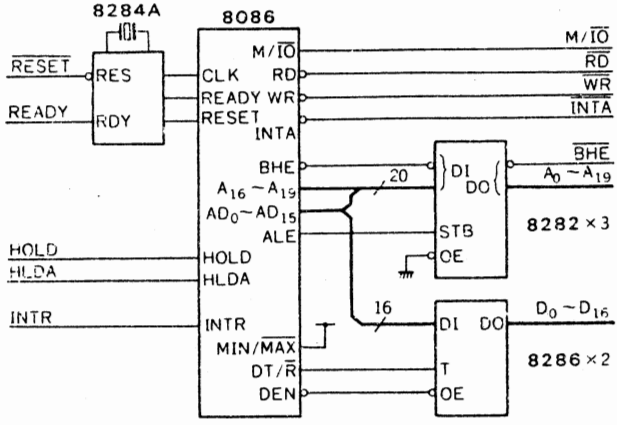


图1.7 8086最小方式的连接举例

图1.9为8086最大方式的连接举例。它与最小方式的最大不同是采用了总线控制器8288及定义了$\overline{RQ}$/$\overline{GT}$和$\overline{LOCK}$信号（参阅4.1）。采用这种结构的目的是为了与8089或8087一起在多处理器系统中使用（参阅1.5）。为了适合于多总线系统，各个命令信号$\overline{MRDC}$（存储器读）、$\overline{MWTC}$（存储器写）、$\overline{IORC}$（输入/输出读）、$\overline{IOWC}$（输入/输出写）是与M/$\overline{IO}$（存储器/输入输

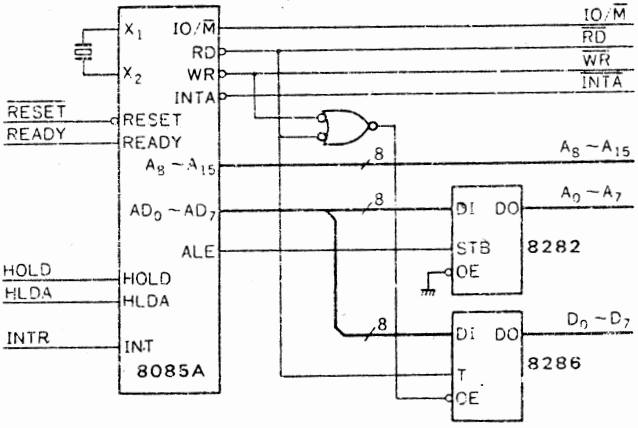


图1.8　8085A系统连接的举例

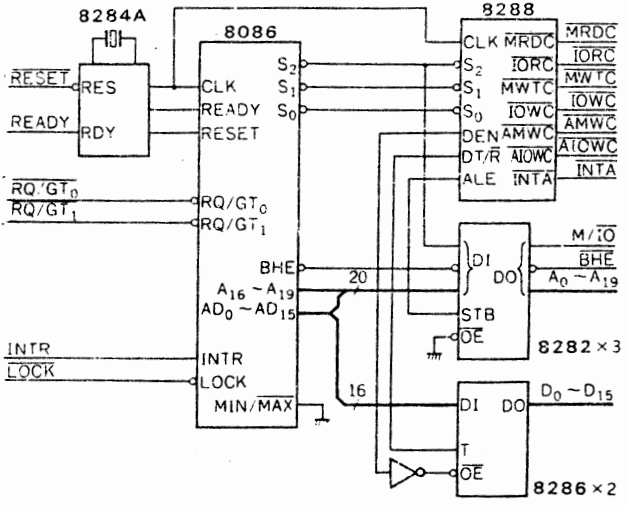


图1.9　8086最大方式的连接举例

出）信号分开的，由8288输出可以直接送到多总线上。而提前的存储器写命令（$\overline{\text{AMWC}}$）、提前的I/O写命令（$\overline{\text{AIOWC}}$）比存储器写命令（$\overline{\text{MWTC}}$）、I/O写命令（$\overline{\text{IOWC}}$）领前1个时钟，因此对外围芯片定时条件是有利的。但必须注意此时读数据并未有效，因此不能用于对多总线输出的信号。

1.2 通过偏移和段指定地址

1.2.1 通过偏移和段指定地址

如上节所述，8086具有1 Mb的存储器地址空间，可是内部寄存器包括指令指针、堆栈指针都只有16位，显然不能直接寻址1 Mb的存储空间。为此，引入了分段的新概念，段为16位。通过偏移（相当于8085A中所使用的地址）和段两个16位值来指定一个存储器地址。

如图1.10所示，8086将段向左移4位与偏移值相加后输出，作为20位物理地址。

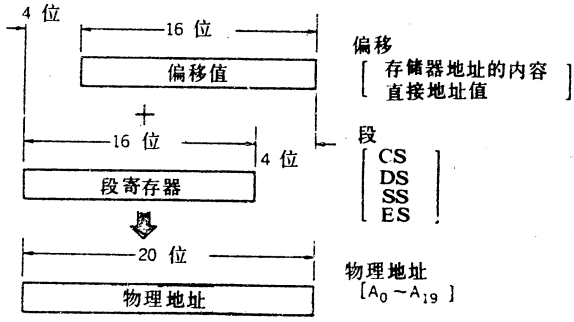


图1.10 物理地址的生成

下面以复位后最初的取指令为例说明地址的生成。取指令时，用指令指针（IP）的内容作为偏移地址，用CS（代码段）寄存器

的内容作为段。由于复位时指令指针的初始值为0000_{16}，CS寄存器的初始值为$FFFF_{16}$，则根据算式：

$$\begin{array}{r} 0000_{16} \\ +\ FFFF_{16} \\ \hline FFFF0_{16} \end{array}$$

可得8086最初的指令是执行物理地址为$FFFF0_{16}$的内容。为此，8086通常将存储器的最高地址部分分配给ROM，在$FFFF0_{16}$地址存放转移类指令。$FFF6_{16}$地址以后规定用于8289初始化以及其他备用。

从上述的说明中可知，在8086中为了处理1 Mb的存储空间，将存储器分成若干个段，每个段的大小为64Kb。分段后的地址共20位，段地址的起始值最低4位为0000_{16}，而高端的16位存放在4个段寄存器CS、DS、SS和ES中。20位字长的真正物理地址是由16位的段地址和16位的偏移地址相加而成的。

1.2.2 段寄存器

在8086中进行段的指定时，采用4个段寄存器的内容。

通常，微处理器在进行存储器存取时有三种情况，对8085A来说，这三种情况采用下面3个寄存器中的内容作为地址值，如图1.11所示。

取指令——程序计数器（PC）

堆栈操作——堆栈指针（SP）

数据处理——HL寄存器（或直接地址）

在8086中，由于用偏移和段指定地址，所以采用了下述6个寄存器，如图1.12所示。

取指令——CS寄存器和指令指针（IP）

堆栈操作——SS寄存器和堆栈指针SP

数据处理——DS寄存器和BX寄存器（或有效地址）

可见，段的指定在取指令、堆栈操作、数据处理三种不同的情况分别采用了CS（代码段）寄存器、SS（堆栈段）寄存器、DS

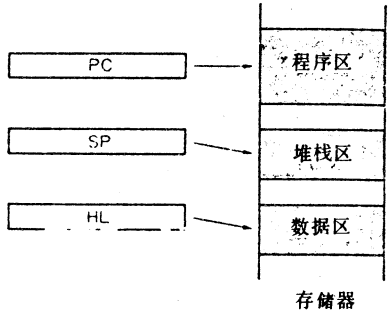


图1.11　8085A的存储器寻址

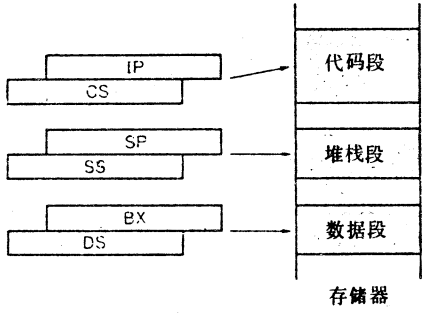


图1.12　8086的存储器寻址

（数据段）寄存器。这样，即使偏移值相同，只要各段寄存器的内容不同，也可以将图1.12所示的指令、堆栈、数据的各区域分配到完全不同的存储器空间。在8086中把通过一个段寄存器所指定的偏移 0000_{16} ～ $FFFF_{16}$ 的64Kb的区域称为段。

用段的概念将1Mb 的存储器空间分成64K个区域，8086中以16字节为单位进行段的更新，各段相互复盖。图1.13表示了8086中存储器空间划分的情况。

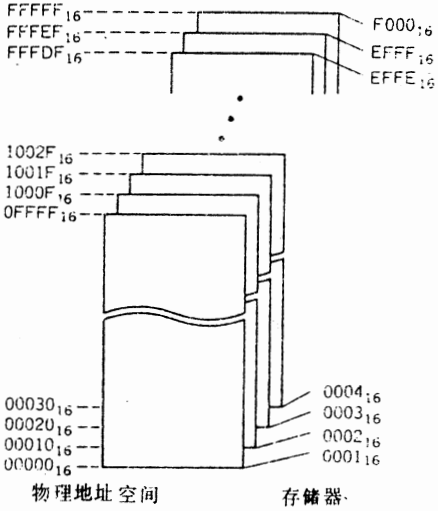


图1.13　存储器空间段的划分

在8086中，还可以通过字符串处理指令对存储器的数据模块进行传送。这时由于需要在1条指令当中指定源和目的两个数据区，因此采用了SI寄存器和DS寄存器指定源数据区，而采用了DI寄存器和ES（附加段）寄存器指定目的数据区。（参阅图1.14）

1.2.3　段寄存器内容的变更

SS寄存器、DS寄存器、ES寄存器的内容可以通过传送指令

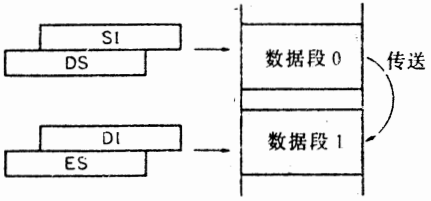


图1.14　字符串传送时的寻址

进行初始化或者进行变更。CS寄存器与上面三个寄存器不同，由于它的内容是地址的一半（另一半为指令指针的内容），所以只能通过JMP、CALL、RET、INT、IRET等指令改变（参阅附录1）。但可将CS寄存器的内容传送给通用寄存器，如下例的最后一条指令。

段寄存器传送指令的举例：

```
MOV  DS,  AX   ;  (DS) ← (AX)
MOV  SS,  BP   ;  (SS) ← (BP)
MOV  ES,  mem  ;  (ES) ← (mem)
MOV  SI,  DS   ;  (SI) ← (DS)
MOV  mem, SS   ;  (mem)←(SS)
MOV  DX,  ES   ;  (DX) ← (ES)
MOV  BX,  CS   ;  (BX) ← (CS)
```

例中（mem）为存储器的内容

1.2.4 段更换

在1.2.2中已叙述了存储器存取时采用段寄存器的情况。在数据区进行存取时，如果在各指令之前插入一个字节的段更换前缀，

表1.4 存储器存取时的缺席段和段更换

存储器存取方式	缺席段	段更换	偏移
取指令	(CS)	不可	(IP)
堆栈操作（CALL、RET、PUSH、POP 指令等）	(SS)	不可	(SP)
数据存取（BP的间接寻址方式除外）	(DS)	ES、CS、SS	EA
数据存取（BP的间接寻址方式时）	(SS)	ES、CS、DS	EA
字符串处理指令（源）	(DS)	ES、CS、SS	(SI)
字符串处理指令（目的）	(ES)	不可	(DI)

（注）EA为有效地址

则只用1条指令就可以使用其他的段寄存器，这种情况通常称为段更换。表1.4列出了各种段更换的情况。在基址指针的间接寻址时，如果不进行段更换，需用SS寄存器，而通过段更换就能指定DS寄存器、CS寄存器和ES寄存器。用汇编语言描述插入段更换前缀时，可通过在各符号之前写上CS:、SS:、DS:、ES:等段名来进行描述。有关基址指针的寻址、汇编语言描述的情况将在1.3中详细说明。

1.2.5 用汇编语言对段、偏移的指定

在8085A的汇编语言中，为了区别存储器地址和数据，采用了立即数助记符。程序例1就是其中的一例。从例中可以看出，在将16位数据装入HL寄存器时，用LHLD助记符装数据，而用LXI装地址。8086与8085A不同，由于8086表示的地址包括段和偏移两部分，所以需要分别用SEG、OFFSET两个算符来指定。程序例2为8086汇编语言进行地址指定的例子。从例中可以看出，无论是段还是偏移都统一采用助记符MOV，而且在助记符上不加任何算符。只要在操作数中指定带SEG、OFFSET算符的变量名，就可以分别装入偏移值和段值。

程序例1 8085A中用汇编语言指定地址

```
0000                    ORG   0
                ;       .
                ;       .
                ;-------------------------------------------------
0000 217856             LXI   H,DATA1 ; (HL) <-- 5678H      (1)
                ;
0003 2A7856             LHLD  DATA1   ; (HL) <-- 1234H      (2)
                ;-------------------------------------------------
                ;       .     数据    地址
                ;
5678                    ORG   5678H   ;                     (3)
                ;
5678 3412       DATA1   DW    1234H   ;                     (4)
```

程序例2 8086中用汇编语言指定地址

```
  0001                  CSEG                                          (1)
                  ;
0000 B8 BC 9A           MOV   AX,9ABCH ;                              (2)
0003 8E D8              MOV   DS,AX    ;                              (3)
                  ;     .
                  ;     .
                  ;-----------------------------------------------------
0005 8B BC 9A           MOV   BX,SEG DATA1     ; (BX) <-- 9ABCH (4)
                  ;
0008 8B 78 56           MOV   BX,OFFSET DATA1  ; (BX) <-- 5678H (5)
                  ;
0008 8B 1E 78 56        MOV   BX,DATA1         ; (BX) <-- 1234H (6)
                  ;-----------------------------------------------------
                  ;     .          数据   段      偏移
                  ;
  9AEC                  DSEG  9ABCH ;                                 (7)
                  ;
                        ORG   5678H ;                                 (8)
                  ;
5678 34 12        DATA1 DW    1234H ;                                 (9)
```

1.3 寻址方式

1.3.1 数据存取的寻址方式

8086微处理器中有下述几种寻址方式。这些寻址方式是用操作（OP）码中的MOD（2位）及R/M（3位）共5位字段来指定的。

（1）寄存器

（2）直接地址

（3）寄存器间接地址

（变址及基址）

（4）基址＋变址

（5）（基址或变址）＋位移

（6）基址＋变址＋位移

这里提到的位移是指在指令的末尾所加的8位或16位的数值。表1.5列出了8086的各种寻址方式。当MOD＝11时，称为寄存器方式，存取通用寄存器。寄存器的指定是由R/M字段进行的，在W＝1时，选择AX～DI寄存器，作为16位寄存器使用；而W＝0时，选择AL～DH寄存器作为8位寄存器使用。当MOD≠11时称为存储器方式，存取存储器。在MOD＝00，R/M＝110时为直接

表1.5 8086 的寻址方式

MOD / R/M	存储器方式			寄存器方式	
	有效地址的计算公式			W=0	W=1
	00	01	10	11	
000	(BX)+(SI)	(BX)+(SI)+D8	(BX)+(SI)+D16	AL	AX
001	(BX)+(DI)	(BX)+(DI)+D8	(BX)+(DI)+D16	CL	CX
010	**(BP)+(SI)**	**(BP)+(SI)+D8**	**(BP)+(SI)+D16**	DL	DX
011	**(BP)+(DI)**	**(BP)+(DI)+D8**	**(BP)+(DI)+D16**	BL	BX
100	(SI)	(SI)+D8	(SI)+D16	AH	SP
101	(DI)	(DI)+D8	(DI)+D16	CH	BP
110	D16（直接地址）	**(BP)+D8**	**(BP)+D16**	DH	SI
111	(BX)	(BX)+D8	(BX)+D16	BH	DI

注：①表中黑体字的寻址方式中，缺席段使用（SS）。

②MOD、R/M、W：在指令中分别包括3、2、1位字段。

D16、D8：16位或8位的位移（即在指令值上增加的16位或8位的值）。

（ ）：表示寄存器的内容。

地址寻址方式，其余均是间接地址寻址方式。当 MOD = 00 时沒有位移，而MOD = 01时有 8 位位移，MOD = 10时有16位位移。在存储器方式时，用R/M指定基址、变址以及基址 + 变址。由于 BX寄存器和基址指针通常用于指定数据区的基地址，所以称为基址寄存器。而SI寄存器、DI 寄存器大多用来表示与基址的偏移值，故称为变址寄存器。不过，无论在基址中还是在变址中，BX、SI、DI 寄存器的功能是完全相同的。基址指针是在堆栈区域内处理数据时为指定基地址而准备的寄存器，在基址指针寻址时，缺席段采用SS寄存器的内容，这一点务必注意。所谓缺席就是不特别指定，即表示不进行段更换的意思。

表1.5中所计算的地址值，在8086中称为有效地址。

假定位移用 2 的补码表示，其值为 8 位时是－128～＋127，其值为16位时是－32768～＋32767。在8086中，如果偏移值超过 65563时，就不再进位，而又返回 0，所以16位的位移看成0～65536和看成－32768～＋32767其结果相同。

1.3.2 立即数据与寄存器间的传送和运算

在1.3.1中并沒有涉及到立即数据，这是因为在 8086 微处理器中立即数据要进行与寻址不同的处理。

在8086中进行运算、传送的源和目的数据还有下述三种情况。（此外，AX、AL寄存器作为累加器接受特殊处理时，有段寄存器的传送指令。）

（1）r/m或R/M（寄存器/存储器）

表1.5中所列32种寻址方式的一种。

（2）reg或REG（通用寄存器）

当W = 1时，为AX～DI寄存器中的某一个。

当W = 0时，为AL～DH寄存器中的某一个。

（3）imm（立即数据）

当W = 1时，为16位数据。

当W = 0时，为 8 位数据。

在上述三者之间可以进行下面的传送和运算：

（1）reg←（reg）○r/m

（2）r/m←(r/m) ○reg

（3）r/m←(r/m) ○imm

注释·传送指令时没有（ ）○。

·○表示运算符号（AND、OR、ADD、SUB等）。

·运算不包括乘除法、ASCII码、BCD码的运算。

（1）和（2）是通用寄存器之间以及通用寄存器与存储器之间的传送和运算。（3）为通用寄存器或存储器与立即数据之间的传送和运算。（2）、（3）中不仅通用寄存器可作累加器使用，存储器也可作累加器使用。这一特点与8085A等微处理器相比是一个很大的不同。

1.3.3 分支转移指令中的寻址

表1.6 分支转移指令的寻方式

指　令	操作数	寻　址	内　容
无条件转移及调用	D8	IP相对	(IP)←(IP)+DB
	D16	IP相对	(IP)←(IP)+D16
	32位直接地址	直接（段外）	(IP)←最初的16位操作数 (CS)←下一个16位操作数
	r/m	间接	(IP)←表1.5所列的存储器或寄存器的内容
	r/m	间接（段外）	(IP)←表1.5所列的存储器内容 (CS)←上述存储器地址+2的存储器内容
条件转移	D8	IP相对	(IP)←(IP)+D8

（注）D8：8位位移（-128～+127）

D16：16位位移（-32768～+32767）

表1.6列出了分支转移指令中的寻址方式。8085A 的所有分支转移指令都只能指定16位的直接（绝对）地址，而8086则基本上是指令指针相对地址，因此程序浮动（可再定位）性好。（这里所讲的浮动性是指能够改变程序在存储器中位置的性质）。在向其他码段进行转移和调用，以及寄存器或存储器间接转移时，采用的是直接（绝对）地址。

条件转移只能在（IP）－128～（IP）＋127的范围内转移。如果希望向该范围之外分支转移时，可按下例的①→②→③的次序执行，即首先在可能转移的范围内转移，然后再通过无条件转移转到目的区。

```
      ⋮
      CMP   AX, DI
      JB    TL1          ①
      ⋮                  127字节以内
      JMP   LT2
TL1:  JMP   TL3          ②
      (到目的程序)       ③
```

程序例 3　相对地址转移和存储器间接地址转移

```
                        CSEG
                        ORG   0
                ;
0000 E9 FD 01           JMP   PROC1      ; DIRECT JUMP              (1)
                ;
0003 2E FF 26 08 00     JMP   CS:PPROC0  ; MEMORY INDIRECT JUMP     (2)
                ;
0008 00 01      PPROC0  DW    PROC0      ; PROCEDURE POINTER        (3)
        以变量定义      ;
                ;       .                由于PPROC1在编码段内
                ;       .                定义，所以需要加段前缀
                ;       ORG   100H
                PROC0:  ;                                           (4)
                ;       .
                ;       .
                ;       .
                        ORG   200H
        以标号定义－    PROC1:  ;                                   (5)
                ;       .
                ;       .
```

在8086微处理器中，相对地址转移和存储器间接地址转移的表示形式相同，这一点必须注意。程序例3是相对地址转移和存储器间接地址转移的例子。例中（1）和（2）的表示形式相同，但是（1）的目标符号PROC1是作为标号（单元）来定义的，因此是相对地址转移。而（2）的目标符号PROCO是以变量来定义的，所以是间接地址转移（详细請参考5.8086的汇编语言）。

1.3.4 用汇编语言描述的实例

程序例4是各种寻址方式用汇编语言描述的实例。

例A部分为寄存器的指定和直接地址的例子，其中（6）是以字节存取寄存器的例子。在8086微处理器中，即使在字节存取时，也可以不限制高位字节、低位字节，进行自由存取。例A中（7）为字存取的例子，由于SP是通用寄存器，因此也可以作累加器。例A中（8）、（9）为直接地址的例子，在（9）中，存储器也可作累加器使用。例A中（10）也是直接地址的例子，但这里的WORD1，因在例3（D）中是以字变量定义的，而AH寄存器要传送的却是一个字节，所以产生错误（ERROR）。

例B部分为立即数和间接寻址方式的例子。其中(11)、（12）为立即数据的传送，由于ADATA在（28）中是以数值定义的，因此只描述符号即可。而WORD0是以变量定义的，它可以通过OFFSET算符产生立即数据。在内容上两者是相同的。例B中（14）、(15)、（16）为间接寻址的例子。间接地址的指定是将地址寄存器用〔 〕括起来进行的。（15）的例子在INTEL公司的MDS上的汇编里也可以按下述形式描述：

```
MOV    CX, [BX] [SI]
```

但现在（15）的描述方法既可以在MDS的汇编中使用，也可以在CP/M的汇编中使用。所以，统一采用这种描述方法使用起来较方便。（16）、（17）为符号位移的表示方法。这两种表示方法结果相同。

例C部分为段更换前缀描述的例子。在（20）、（21）中，由于

程序例4　用汇编语言描述寻址方式的举例

```
                        CSEG            ; DS寄存器、ES寄存器的初始化   (1)
0000 B8 00 20           MOV   AX,2000H  ;                              (2)
0003 8E D8              MOV   DS,AX     ;                              (3)
0005 B8 00 30           MOV   AX,3000H  ;                              (4)
0008 8E C0              MOV   ES,AX     ;                              (5)
                        ;
                        ; (A) ------------------------------------------------
000A 8A C6              MOV   AL,DH     ; (AL) <-- (DH)                (6)
000C 03 E7              ADD   SP,DI     ; (SP) <-- (SP)+(DI)           (7)
000E 8B 36 00 10        MOV   SI,WORD0  ; (SI) <-- (WORD1)             (8)
0012 28 2E 04 10        SUB   BYTE0,CH  ; (BYTE0) <-- (BYTE0)-(CH)     (9)
                        ;
** ERROR NO:  8 **  NEAR: ";*   OPERAND(S) MISMATCH INSTRUCTION
0016 90 90 90 90 90 90  MOV   AH,WORD1  ; ERROR                        (10)
                        ;   因向字节寄存器装入字变量而产生错误
                        ;
                        ;
                        ; (B) ------------------------------------------------
001C BB 00 10           MOV   BX,ADATA          ; (BX) <-- 1000H       (11)
001F BB 00 10           MOV   BX,OFFSET WORD0   ; (BX) <-- 1000H       (12)
                        ;                          [ ] 表示间接地址
                        ;
0022 BE 02 00           MOV   SI,2              ; (SI) <-- 2           (13)
0025 8B 07              MOV   AX,[BX]           ; (AX) <-- (WORD0)     (14)
0027 8B 08              MOV   CX,[BX+SI]        ; (CX) <-- (WORD1)     (15)
0029 8A 50 03           MOV   DL,[BX+SI]+3      ; (DL) <-- (BYTE1)     (16)
                        ;
002C 8B 85 00 10        MOV   AX,[DI]+ADATA     ; (AX) <-- (WORD1)     (16)
0030 8B 85 00 10        MOV   AX,ADATA[DI]      ; (AX) <-- (WORD1)     (17)
                        ;
                        ; (C) ------------------------------------------------
0034 BD 00 10           MOV   BP,ADATA          ; (BP) <-- (1000H)     (19)
0037 2E A1 4A 00        MOV   AX,CS:CDATA       ; (AX) <-- (CDATA)     (20)
003B 26 A1 00 00        MOV   AX,ES:EDATA       ; (AX) <-- (EDATA)     (21)
003F 3E 8B 02           MOV   AX,DS:[BP+SI]     ; (AX) <-- (WORD1)     (22)
                        ;                          段前缀的指定
0042 2E A1 4A 00        MOV   AX,CDATA          ; (AX) <-- (CDATA)     (23)
0046 26 A1 00 00        MOV   AX,EDATA          ; (AX) <-- (EDATA)     (24)
                        ;
004A 00 00              CDATA   DW    0                                (25)
                        ;
                        ; (D) ------------------------------------------------
                        ;
  2000                          DSEG  2000H     ;                      (26)
                                ORG   1000H     ;                      (27)
                        ;
  1000                  ADATA   EQU   OFFSET $  ;                      (28)
1000 00 00              WORD0   DW    0 ; WORD DATA                    (29)
1002 00 00              WORD1   DW    0 ; WORD DATA                    (30)
1004 00                 BYTE0   DB    0 ; BYTE DATA                    (31)
1005 00                 BYTE1   DB    0 ; BYTE DATA                    (32)
                        ;
  3000                          ESEG  3000H     ;                      (33)
                                ORG   0         ;                      (34)
0000 00 00              EDATA   DW    0         ;                      (35)
```

段前缀指令

CDATA、EDATA分别是由代码段、附加段定义的，所以需要有段更换前缀。例C（22）为用BP有关的间接地址存取数据段的例子（如果什么也不指定，SS寄存器的内容作为段使用）。（20）、（21）、（22）中各操作码前面的2E、26、3E就是段更换前缀指令。实际上，除了（22）的间接地址外，变量段是由汇编程序本身进行检查，并自动插入段更换前缀。所以，（20）、（21）象（23）、（24）那样描述也会得到同样的结果。

1.4 中　断

在8086微处理器中有256级中断。在256级中断中以0～255的序号表示中断的类型。序号越小优先级越高。

在256级中断中，类型0～4已在8086内部进行了定义。类型5～31是INTEL公司为将来的支援软件、芯片而准备的。类型32以上的中断是用户可以使用的。类型224是用于CP/M—86系统调用的软件中断。

1.4.1 中断的种类

表1.7列出了8086微处理器的中断种类。其中NMI（非屏蔽中断）和硬件中断为外部（硬件）中断.其余的中断在8086内部产生。

（1）0作除数中断

在执行除法指令时，如果商超过8位或16位所能表达的最大值（如除以0时），就无条件产生这种中断，并具有最高优先级。

（2）单步中断

它是在监控程序中，为了进行单步调试而提供的中断形式。设定TF（陷阱标志）后，如执行下一条指令，就会产生这种中断。陷阱标志的设定方法請参考5.5.1。

（3）NMI

为非屏蔽的外部中断，不进入INTA周期（参考1.4.2）。这种中断在外部中断中具有最高的优先级。通常，在电源断、存储器奇

表1.7　中断的种类

种　类	类型	优先级	IF屏蔽	产 生 的 原 因	转移所需时钟数
0作除数	0	1	不可	除的结果超过所能表达的最大值时	
单步	1	2	不可	TF置1，执行下一条指令后	50
NMI	2	3	不可	NMI端检出高电平时（边缘检测）	50
1字节	3	4	不可	执行INT 3指令时	52
溢出	4	5	不可	OF为“1”时，执行INTO指令时	53
软件	0～255	6	不可	执行INTn指令（n＝0～255）时	51
硬件	0～255	7	可	在INTR端检出高电平时（电平检测）	61

（注）①NMI：非屏蔽中断；IF：中断允许标志；TF：陷阱标志；OF：溢出标志

②硬件中断的类型在第二个INTA周期时，从D_7～D_0输入。

③硬件中断时，类型越小优先级越高。

偶校验出错等情况时使用。为了可靠地产生中断，先将NMI端保持2个时钟周期以上的低电平，然后再提高电平，并维持2个时钟周期的高电平。

（4）1字节中断

这种中断是为设置软件断点而准备的。当执行INT 3指令时，就产生这种中断。它的功能与软件中断相同，但是为了便于与其他指令置换，INT 3指令是1字节指令。

（5）溢出中断

在设置OF（溢出标志）时，如果执行INT C指令就产生溢出

中断。这种中断是为了检测带符号数据的运算结果是否出现溢出。

（6）软件中断

当执行INTn指令（n为类型n = 0～255）时就产生软件中断。INTn指令相当于8085A中的RSTn指令。不过在8085A中是在8×n地址存放转移指令，而在8086中是在4×n地址存放分支转移的偏移和段。

（7）硬件中断

这种中断是普通的外部中断。只有这种中断才能用IF（中断允许标志）屏蔽。8086在各指令的最后时钟周期（不是T_4）对INTR端采样，如果是高电平，就进入INTA周期。在$\overline{\text{INTA}}$信号输出之前，INTR端必须保持高电平。只有在向段寄存器执行MOV、POP指令以后，INTR端才不采样（其他中断也同样不采样）。

1.4.2 接受中断的顺序

图1.15表示8086从检出中断请求到转移到中断处理程序的流程图。

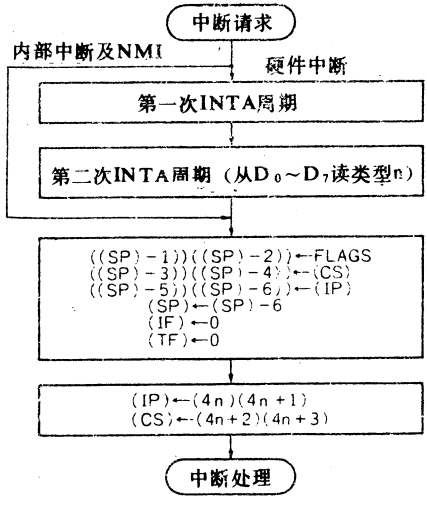


图1.15 接受中断请求后转移到中断处理程序的流程

如果是硬件中断，8086就连续执行2个INTA周期。第1个INTA周期是为级联连接8259A时而准备的，在这一周期中，主中断控制器8259A向从中断控制器输出级联地址。8086在第1个INTA周期中不读也不写。在第1个INTA周期中$\overline{LOCK}$信号是有效的（低电平）。第2个INTA周期是8086读类型n的周期，从D_7～D_0读取8259A输出的类型向量（8位），因此8259A必须分配在偶数地址。

INTA周期结束之后，或者在产生内部中断、NMI中断时，8086微处理器就将标志、CS寄存器、指令指针的内容保留在堆栈中，清除中断允许标志和陷阱标志。并且将存储器的4n和4n+1地址的内容及4n+2和4n+3地址的内容分别作为新的指令指针及CS寄存器的内容装入指令指针和CS寄存器，后转到中断处理程序。

存储器的0_{16}～$3FF_{16}$（=4×256−1）地址作为中断处理程序的指针表使用（参考图1.16）。通常，这个区域在RAM中，通

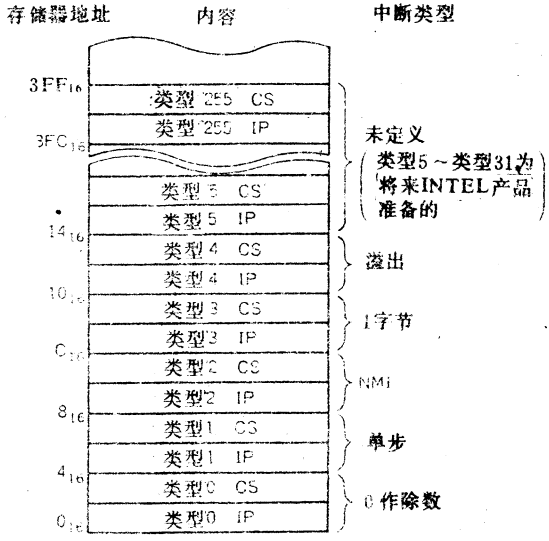


图1.16 中断向量表

过程序写入中断处理程序的段和偏移值。在8086微处理器中，只有这个中断指针表和复位后开始执行的地址 $FFFF0_{16}$～$FFFFF_{16}$ 是接受特殊处理的存储区区域。

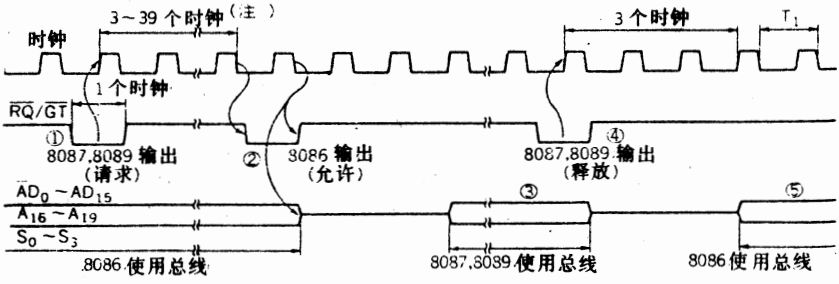
1.5 多处理器系统

为了更容易地组成多处理器系统，8086准备了下述三种结构，这三者都只在最大方式时采用。

1.5.1 $\overline{RQ}/\overline{GT}$（請求/允许）信号

$\overline{RQ}/\overline{GT}$是为裁决 8087、8089 和总线使用权的信号。该信号为漏极开路的双向性信号，与8087、8089以交换方式进行裁决。

图1.17示出了$\overline{RQ}/\overline{GT}$信号的波形图。通常，8086 在使用总线的状态下，$\overline{RQ}/\overline{GT}$ 为高电平（准确地说是浮动状态）。在这种状态下，当8087或8089需要存取总线时，8087、8089就要在 1 个时钟区间使$\overline{RQ}/\overline{GT}$处于低电平（图1.17中的①状态）。8086检出该請求信号，如果总线处于可以开放的状态，则$\overline{RQ}/\overline{GT}$变为低电平(图中②状态）作为允许信号。这时8087、8089检出该允许信号，对总线进行存取（图中③状态）。8087、8089在使用总线结束之后，再将$\overline{RQ}/\overline{GT}$信号变成低电平（图中④状态）。8086再检出该信号，又开始对总线的存取（图中⑤状态）。



图中

（注）从请求信号输入到允许信号输出的潜伏时间如下：

执行中的指令	潜伏时间
无LOCK的指令执行中	3～6（10）时钟
INTA周期中（$\overline{\text{LOCK}}$有效）	15 时钟
LOCK XCHG指令执行中	24～29（39）时钟

（ ）内的值为存取从奇数地址开始的字时

图1.17 $\overline{RQ}/\overline{GT}$信号的波形

以上说明了仲裁功能的概要。在8086芯片上有两个$\overline{RQ}/\overline{GT}$端，即$\overline{RQ}/\overline{GT_0}$和$\overline{RQ}/\overline{GT_1}$。8087、8089两个都可连接。两个$\overline{RQ}/\overline{GT}$信号具有完全相同的功能，但是$\overline{RQ}/\overline{GT_0}$的优先级高，在二者同时有請求信号输入时，在$\overline{RQ}/\overline{GT_0}$端上输出允许信号。

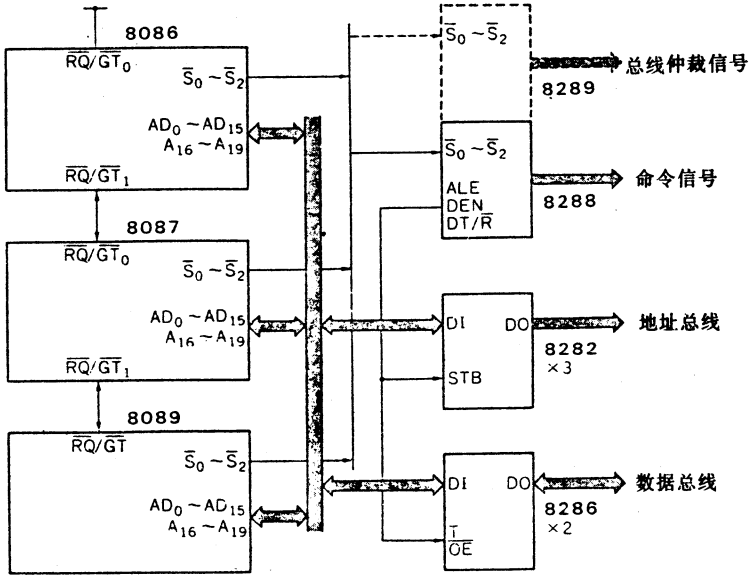


图1.18 由8086、8087、8089组成的多处理器系统

图1.18为8086、8087、8089组成系统的例子。在8087上有两个可以级联的$\overline{RQ}/\overline{GT}$端，8086、8087、8089连接如图1.18所示。状态信号S_0~S_2也共同连在一起，有效的处理器状态信号输出到8288、8289。这是因为在最大方式时，命令信号是从另外的芯片8288输出的。8086的$\overline{RQ}/\overline{GT}_0$端处于高电平状态，该端还可以连接8089。

1.5.2 8289总线仲裁器

$\overline{RQ}/\overline{GT}$信号是为了对处理器芯片间进行仲裁而设置的信号，而8289是在处理器模板间进行仲裁的芯片。使用8289的示意图如图1.19所示。

通常8086与I/O处理器8089组成的多处理器系统按图1.18所示的组成。但是，如果两个处理器频繁地存取总线时，处理器的等待时间就要加长，使执行的速度减慢。为了解决这个问题，采取了图1.19的方法，将8086和8089分装在不同的模板上。在通常的情况下，存取本地总线上的存储器，只在需要时才存取系统总线（多总线）上的存储器。这样一来，就使得两个处理器同时存取系统总线的机会减少了，从而提高了系统的处理能力，增加了系统的吞吐量。由于$\overline{RQ}/\overline{GT}$不能用于8086与8086之间的连接，所以在使用二个以上的8086时，也要采取图1.19所示的系统组成。（即使连接两个8086的$\overline{RQ}/\overline{GT}$端，由于二者都作为主处理器工作，复位后，会同时去存取总线。）

1.5.3 $\overline{LOCK}$信号

$\overline{LOCK}$信号是组成多处理器系统时采用的信号，如果在指令之前增加1个字节的LOCK前缀，8086在总线存取过程中，$\overline{LOCK}$端一直保持低电平。对多处理器系统来说，在信号变量的检查中，某一个处理器正在执行存储器存取指令时，不允许其他处理器存取同一地址的存储器内容，$\overline{LOCK}$信号就可以实现这一作用。在硬件设计上采用当$\overline{LOCK}$信号有效时，其他处理器不能得到总线的方法。如用总线仲裁器8289时，只要将8086的$\overline{LOCK}$输出连接到8289

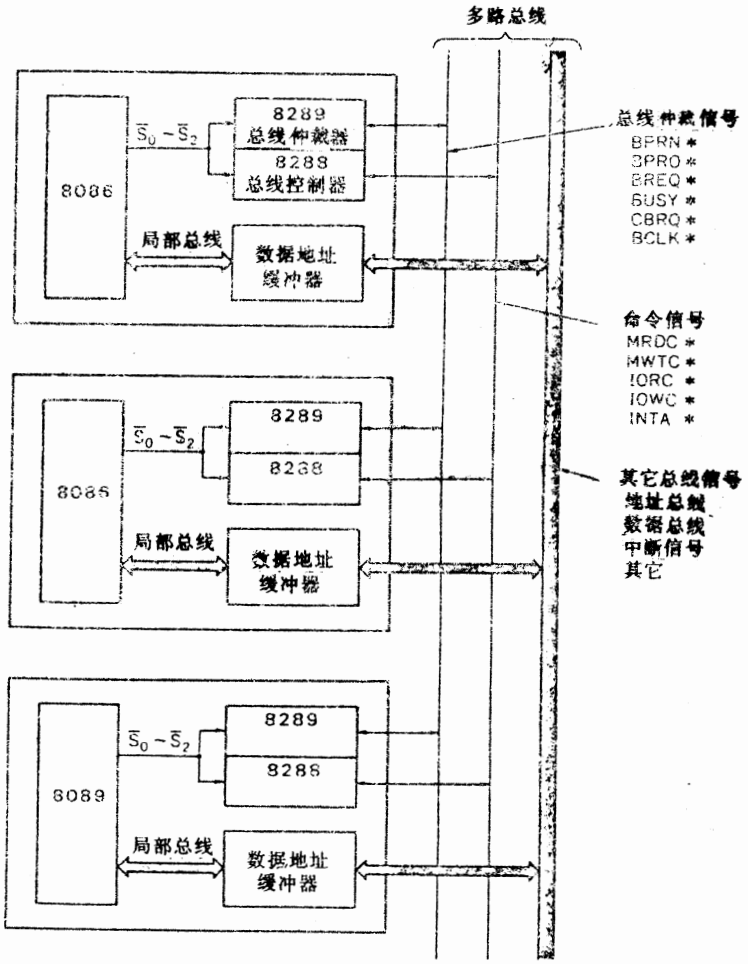


图1.19　多总线上的多处理器系统

的$\overline{LOCK}$输入即可，以后就由8289来控制了。

从下面的例子中可以看出$\overline{LOCK}$信号是很必要的。比如在两个处理器共用一台打印机的系统中，如果两个处理器无秩序地访问打印机，就会出现所打印的数据是两个处理器交替输出的字符。为了防止这种现象，通常在存储器中设置1个字节的标志（FLAG），并定义为：当FLAG＝0时，打印机处于“打开”状态，而当FLAG≠0时，打印机在“使用中”。在处理器访问打印机时，如果FLAG≠0，就要等待，直到FLAG＝0；如果FLAG＝0，首先使FLAG≠0之后再向打印机输出，输出结束后，使FLAG＝0。这种方法称为测试置位法。下面是测试置位法程序的例子。

```
FLAG    DB  0
TESTF:  MOV     AL, OFFH  ; 得到原FLAG内容,并
LOCK    XCHG    AL, FLAG  ; 把FLAG设置为FFH
        AND     AL, AL    ; FLAG=0?
        JNZ     TESTF     ; 如果不是，继续测试
        ⋮
        CALL    PRINT     ; 打印输出
        ⋮
        MOV     FLAG, 0   ; FLAG复位
```

用一条XCHG指令进行FLAG的读取和置位，实现AL寄存器的内容和FLAG内容的交换。因此，当FLAG＝0时，FLAG置位为FF_{16}，如果FLAG≠FF_{16}，就相当于什么也没做。

两个处理器在执行测试置位时的过程如下：

处理器0的执行过程

```
TESTF:  MOV     AL,     OFFH
LOCK    XCHG    AL,     FLAG
        AND     AL,     AL
        JNZ     TESTF
```

处理器1的执行过程

```
TESTF:  MOV     AL, OFFH
LOCK    XCHG    AL, FLAG
        AND     AL, AL
        JNZ     TESTF
```

XCHG指令对存储器来说，首先是读、然后是写。如果在XCHG

指令中不带LOCK前缀，就会产生如图1.20所示的进行测试置位的现象。这时，两个处理器同时读取FLAG“原存”内容，如果FLAG = 0时，两者都要进行打印输出。

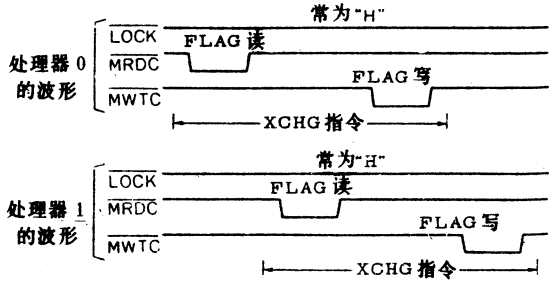


图1.20　不使用$\overline{\text{LOCK}}$信号测试置位的波形图

采用$\overline{\text{LOCK}}$信号就可以解决这个问题，在XCHG指令中如果加上LOCK前缀，可以得到图1.21的波形。这时，$\overline{\text{LOCK}}$信号为低电平，在读写期间，其他的处理器就不能存取总线。这样就避免了两个处理器同时读取“原存”FLAG内容的情况。

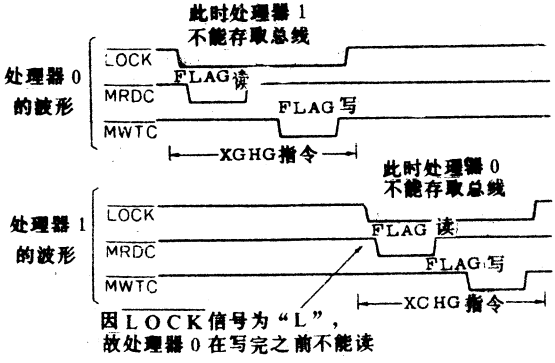


图1.21　采用$\overline{\text{LOCK}}$信号测试置位的波形

1.6　执行部分和总线接口部分

8086内部结构分为执行部件(EU)和总线接口部件（BIU）两部

分，如图1.22所示。执行部分由ALU(运算器)、通用寄存器和标志组成，进行16位的各种运算及有效地址的计算。总线接口部分负责与存储器、I/O的接口，由段寄存器、指令指针、地址加法 器和指令队列缓冲器组成。地址加法器将段和偏移的内容相加，生成20位的物理地址。

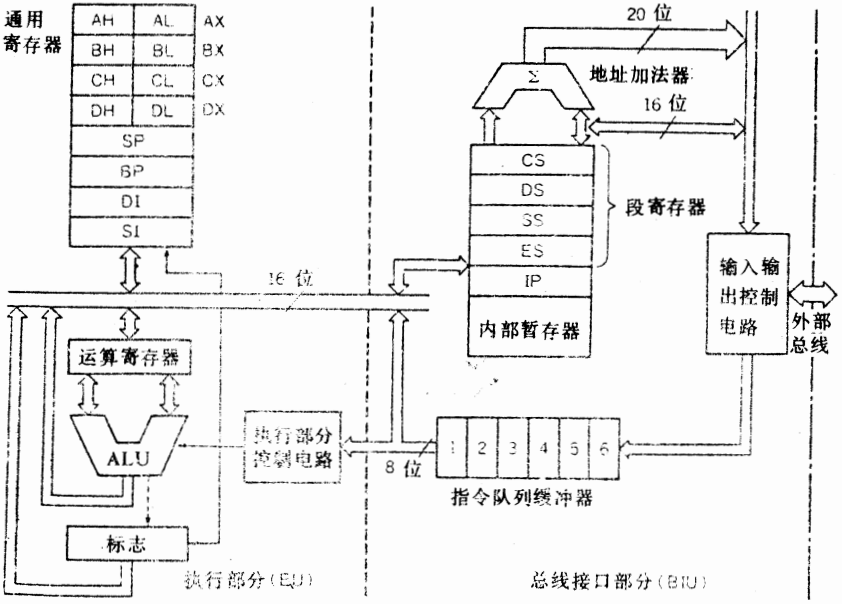


图1.22　8086的内部结构框图

总线接口部分的动作与执行部分是完全不同步的。通常，将指令先读入指令队列缓冲器中，如果执行部分有对存储器和I/O的存取請求时，在执行中的存储器周期结束后的下一存储器周期，存取指定的存储器和I/O。指令队列缓冲器为6个字节，当指令队列缓冲器的内容已满，而且执行部分沒有存取請求时，总线接口部分就进入空闲状态。在执行转移、调用、返回指令时，指令队列缓冲器的内容就被清除。

2 8086 的指令

本章将说明8086微处理器指令的概要。文中所提到的指令分类是本书作者自己的分类方法，与INTEL公司的分类方法略有不同。8086的指令系统与8085A相比，在下述几方面有较大的扩充。

- 有8个通用寄存器可作为累加器使用。
- 字节、字的处理同样进行。
- 有重复指令和乘除运算指令。
- 扩充了的条件转移、移位/循环指令。
- 可进行带符号的运算。

2.1 指令编码

图2.1表示了8086指令编码的基本格式。8086的指令编码可以分为三大类，在指令编码的后面，如果需要还可以按位移（或直接地址）、立即数据的顺序增加8位或16位的编码。

在以后的说明中，将采用下面符号，其意义是：

（1）r/m或R/M（寄存器/存储器）

表示在MOD，R/M字段中指定寻址方式的通用寄存器或存储器（参考1,3）。

（2）reg或REG（通用寄存器）

表示在REG字段所指定的寄存器。当W＝1时，是AX～DI寄存器，而W＝0时，为AL～DH寄存器（参阅1.1.4）。

（3）sreg（段寄存器）

表示在传送指令中，由SREG字段指定的段寄存器（参考1.2.2）。

（4）acc（累加器）

在W = 1时表示是AX寄存器，W = 0 时表示为AL寄存器。（在8086中，有将AL、AX寄存器作为累加器的特殊处理指令）。

（5）imm（立即数据）

表示立即数据。当W = 1时，为16位数据，而W = 0时为8位数据。

2.1.1 寄存器、存储器的存取指令（I）

这类指令中的操作数是采用通用寄存器（或段寄存器）和寄存器/存储器。比如传送指令、二项运算指令就是这种指令。根据D值不同决定哪一个是目的寄存器，D = 0时，寄存器/存储器为目的寄存器，而D = 1时，通用寄存器为目的寄存器，参阅图2.1的(1)。

在寄存器/存储器为寄存器方式（mod = 11）时，如下例所示，同样功能的指令中存在两种指令编码，这一点应特别注意。至于助记符变换为哪一种编码，依汇编程序而定。

〔例〕ADD BX、CX的指令编码

①reg＝BX，r/m＝CX时

```
00000011    11011001
              BX CX
            D
            ↓
(reg)←(reg)+(r/m)
```

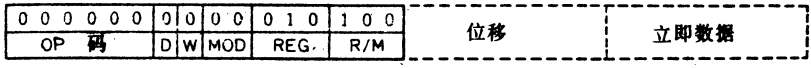
②reg＝CX，r/m＝BX时

```
00000001    11001011
              CX BX
          D
          ↓
(r/m)←(r/m)+(reg)
```

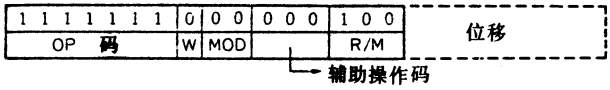
2.1.2 寄存器、存储器的存取指令（II）

操作数为寄存器/存储器的指令。单项运算、移位、循环指令都属于这种指令，参阅图2.1的（2）。

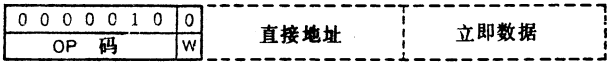
(1) 寄存器、存储器存取指令 I



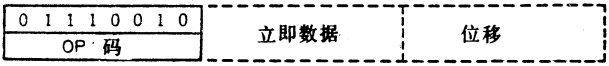
(2) 寄存器、存储器存取指令 II



(3) AL,AX寄存器存取指令



(4) 其它指令



图中：

(注2) MOD, R/M

按表1.5的寻址方式

(注3) W

W=0 字节处理

W=1 字处理

(注4) D

D=0 R/M为目的

D=1 REG为目的

(立即数据的有关指令中无D位)

(注1) REG 寄存器号

REG	通用寄存器	
	W=0	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

图2.1 8086的指令格式

2.1.3 AL、AX寄存器的有关指令

这些指令是为了保持与8085A兼容而准备的。AL、AX寄存器指定作为累加器。有传送、二项运算指令。虽然在寄存器、存储器的存取指令(I)中也有与此功能相同的指令编码，但AL、AX寄存器的存取如采用这种方式，指令编码比较短（汇编程序将会自动变换到这种编码），参见图2.1的（3）。

2.1.4 其他指令

所谓其他指令是指上述三种指令以外的指令，基本上具有8位操作码（其中也有的指令有REG字段）。属于这种指令的有分支转移指令、标志操作指令等。参阅图2.1的（4）。

2.2 传送指令

传送指令可分为下述几类：

- 一般传送指令
- 交换指令
- 堆栈操作指令
- 其他

2.2.1 一般传送指令

这种指令用助记符MOV表示，是普通的传送指令，指令类型如表2.1所示。表中上面四种为基本指令，用于通用寄存器、寄存

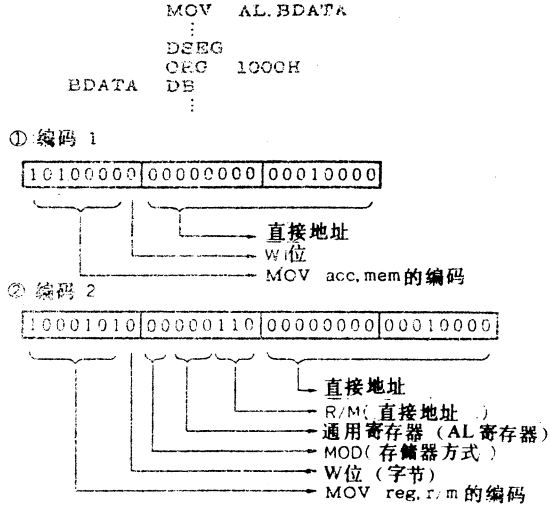


图2.2 两种指令编码在汇编语言中表示同一指令的例子

器/存储器、立即数据之间的传送。AX、AL的有关指令是为了保持与8086A有兼容性而准备的指令。8086的指令系统具有很强的扩充性，相同功能（用汇编语言描述时形式相同）的指令，有的却具有互不相同的指令编码。比如，在图2.2中，将存储器的内容向AL寄存器传送时，把AL寄存器看成acc，还是把AL寄存器看成reg结果是不同的，会得到如例中所述两种不同的指令编码。汇编的结果究竟是哪一种，将依赖于汇编程序，但在通常情况下，前者在汇编中生成的编码较短。在通用寄存器、寄存器/存储器的指令之间也有相同功能的指令。表中下面二条指令是有关段寄存器的指令。有关段寄存器的指令，除了LDS、LES、PUSH、POP以外，只有这一条传送指令。应该注意在段寄存器中不能装入立即数据，只可以从存储器向段寄存器进行传送。

表2.1 MOV 指 令 的 类 型

指　　令	目的寄存器	源 寄 存 器
MOV reg, r/m	通用寄存器	寄存器/存储器
MOV r/m, reg	寄存器/存储器	通用寄存器
MOV reg, imm	通用寄存器	立即数据
MOV r/m, imm	寄存器/存储器	立即数据
MOV acc, mem	AX或AL	存储器（直接地址）
MOV mem, acc	存储器（直接地址）	AX或AL
MOV sreg, r/m	段寄存器	寄存器/存储器
MOV r/m, sreg	寄存器/存储器	段寄存器

2.2.2 交换指令

XCHG指令，对8086微处理器来说，不仅在寄存器—寄存器之间可以进行交换，而且在寄存器—存储器之间也可以进行交换。寄存器—存储器之间的交换，经常采用1.5中讲的测试置位方法。交换指令有下面两种类型。

（1）XCHG reg，r/m

这是通用寄存器与寄存器/存储器间的交换。

（2）XCHG acc，reg

这是AL或AX寄存器与通用寄存器之间的交换。

2.2.3 堆栈操作指令

这是堆栈的进栈、出栈指令。在8086微处理器中，不仅内部寄存器的内容可以进栈、出栈，而且存储器的内容也可以进栈、出栈。标志的进栈、出栈采用PUSHF、POPF助记符。堆栈操作指令的类型如下：

（1）PUSH reg；POP reg

通用寄存器的进栈、出栈。

（2）PUSH r/m；POP r/m

寄存器/存储器的进栈、出栈。

（3）PUSH sreg；POP sreg

段寄存器的进栈、出栈。

（4）PUSHF；POPF

标志的进栈、出栈。

2.2.4 其他传送指令

在8086微处理器中，除了上述的传送指令之外，为了编程方便，还备有下面几种传送指令。

（1）LEA reg，r/m

这是有效地址的计算指令。采用这条指令，就可以不必用软件计算有效地址了。这条指令的意义是由r/m指定的执行地址，装入reg所表示的通用寄存器中。在r/m指定寄存器时，装入的内容不固定。

（2）LDS reg，r/m；LES reg，r/m

这是向DS寄存器、ES寄存器装入的指令。将r/m指定的地址开始的连续4个字节，装入通用寄存器（低端2字节）和DS或ES寄存器（高端2字节）中。

（3）LAHF；SAHF

这是标志的低8位和AH寄存器间的传送指令。LAHF是将标志的内容装入AH寄存器，SAHF是把AH寄存器的内容存入标志中。各标志与AH寄存器的对应关系如下：

AH寄存器	标　志
第0位	进位标志
第1位	*
第2位	奇偶校验标志
第3位	*
第4位	辅助进位标志
第5位	*
第6位	零标志
第7位	符号标志

* 表示在LAHF指令中不固定

（4）XLAT

这是8位的查表字节换码指令。BX寄存器的内容加上AL寄存器的值（0～255）作为存储器的地址，并将该地址的存储器内容装入AL寄存器。其结果破坏了以前AL寄存器的内容。

如果在BX寄存器中装入了换码表的开始地址，就可以进行8位数据的查表换码操作。

2.3 输入输出指令

8086微处理器只有在AL或AX寄存器与输入/输出口之间进行传送的指令。这里的输入/输出口地址包括直接地址和DX寄存器间接地址两种情况。

2.3.1 直接地址输入/输出

这种指令有下述四种类型。这里值得注意的是port指定为8

位，而A_{15}～A_8输出为0。

```
IN    AL,   port   (字节输入)
IN    AX,   port   (字输入)
OUT   port, AL     (字节输出)
OUT   port, AX     (字输出)
```

2.3.2 DX寄存器间接地址输入/输出

这种指令包括下述四种类型：

```
IN    AL,   DX     (字节输入)
IN    AX,   DX     (字输入)
OUT   DX,   AL     (字节输出)
OUT   DX,   AX     (字输出)
```

这种指令中，DX寄存器的内容作为地址输出给A_{15}～A_0。如果象8085A那样使用几个相同的I/O控制器时，就可以通过这条指令，用一个控制程序即可（参考5.5.2的例子）。

2.4 运算指令

表2.2列出了各种运算指令和标志的变化情况。运算指令可分成二项运算、单项运算和校正三种指令。二项运算指令又分为算术运算指令和逻辑运算指令。乘除法指令虽然也是算术运算指令，但由于它的寻址方式与其他二项运算指令不同，因此将另行说明。

在算术运算指令中，乘除运算指令有带符号运算和无符号运算的区别，而加减运算指令却没有这个区别。这是因为，带符号的数据用2的补码表示时，对加减法来说，用无符号运算指令能够得到正确的结果，而对乘除法来说却得到错误的结果，因此，带符号与无符号的乘除运算应使用不同的指令。

表2.3列出了8086微处理器能够处理的数据类型。基本数据是8位、16位无符号及带符号的二进制数。通过在运算结果中执行校正指令，可将8位无符号的二进制数按压缩型十进制数处理，16位无符号二进制数按非压缩型十进制数处理。图2.3为这些十进制数的

表示方法的例子。对于乘除运算指令，在数据扩充之后可进行下述运算。

- 8位乘法　　8位×8位=16位
- 16位乘法　　16位×16位=32位
- 8位除法　　16位÷8位=8位
- 16位除法　　32位÷16位=16位

表2.2　运算指令与标志的变化

种类		助记符	内容	标志的变化 O S Z A P C
二项运算	算术运算	ADD	加　法	○○○○○○
		ADC	带进位的加法	○○○○○○
		SUB	减　法	○○○○○○
		SBB	带借位的减法	○○○○○○
		CMP	减法，但寄存器不变化，只是标志变化	○○○○○○
	逻辑运算	AND	与	0○○△○0
		OR	或	0○○△○0
		XOR	异或	0○○△○0
		TEST	“与”，寄存器不变化，只有标志变化	0○○△○0
	乘除运算	MUL	无符号数的乘法	○△△△△○
		IMUL	带符号的整数乘法	○△△△△○
		DIV	无符号数的除法	△△△△△△
		IDIV	带符号的整数除法	△△△△△△
单项运算		INC	加1	○○○○○×
		DEC	减1	○○○○○×
		NEG	求补码	○○○○○1
		NOT	求反码	××××××

种	类	助记符	内 容	标志的变化 O S Z A P C
校正	十进制校正	DAA	加法的十进制校正	△○○○○○
		DAS	减法的十进制校正	△○○○○○
		AAA	加法的非压缩形式十进制校正	△△△○△○
		AAS	减法的非压缩形式十进制校正	△△△○△○
		AAM	乘法的非压缩形式十进制校正	△○○△○△
		AAD	除法的非压缩形式十进制校正	△○○△○△
	扩充	CBW	变换字节为字	××××××
		CWD	变换字为双字	××××××

表中 C：进位标志　　○：根据结果不同，变为0或1
P：奇偶校验标志　　X：不变化
A：辅助进位标志　　△：不固定
Z：零标志　　0：复位成0
S：符号标志　　1：设置为1
O：溢出标志

8086微处理器不能处理位数据。因此，在检查位的内容时，需要用AND、OR 指令进行屏蔽。无符号、带符号的32位二进制数和浮点运算是由辅助处理器8087进行的。8087除了可以进行80位浮点数据处理外，还可以进行平方根、三角函数等计算。

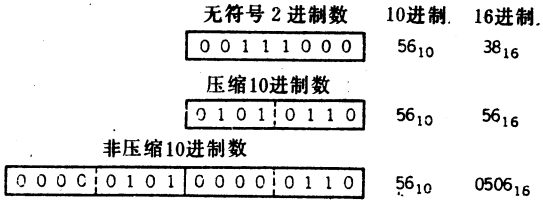


图2.3 十进制数的表示方法

表2.3 8086可以处理的数据类型

数据类型	数据范围	运算	可作累加器使用的寄存器或存储器
无符号二进制（8位）	0～255	加减 乘除	所有的通用寄存器或存储器 AX寄存器
无符号二进制（16位）	0～65535	加减 乘除	所有的通用寄存器或存储器 AX寄存器、DX寄存器
带符号二进制（8位）	-128～+127	加减 乘除	所有的通用寄存器或存储器 AX寄存器
带符号二进制（16位）	-32768～+32767	加减 乘除	所有的通用寄存器或存储器 AX寄存器、DX寄存器
压缩形十进制（8位）	0～99	加减	AL寄存器
非压缩型十进制（8位）	0～99	加减 乘除	AL寄存器 AX寄存器

（注）乘法的积、除法的商的数据位数为表中数据二倍，如8位二进制数与8位二进制数之积为16位。

2.4.1 二项运算指令（乘除法除外）

这种运算指令的形式为：

X←X○Y　　○：算符

不能采用Z←X○Y的形式。

算术运算包括加减运算和比较指令，逻辑运算包括“与”、“或”、“异或”和测试指令。TEST指令与AND指令运算相同，CPM指令与SUB指令运算相同。只是执行TEST指令和CPM指令时，寄存器不变，仅标志变化。

运算指令有下述几种类型。在8086微处理器中，运算指令和MOV指令一样，将目的操作数写在前面，源操作数写在后面。这里仅以ADD指令为例说明，除了TEST指令之外，所有的指令都可

使用这种形式。

（1）ADD reg, r/m

源操作数为寄存器/存储器，目的操作数为通用寄存器。

（2）ADD r/m, reg

与（1）刚好相反，寄存器/存储器为目的操作数，通用寄存器为源操作数。由于在TEST指令中，没有源操作数与目的操作数的区别，虽然在汇编语言中有这一条指令，但却变换成与（1）相同的编码。

（3）ADD r/m, imm

源操作数是立即数据、目的操作数是寄存器/存储器。

（4）ADD acc, imm

源操作数为立即数据、目的操作数为AL寄存器或AX寄存器。在（3）中也有与（4）相同功能的指令，但（4）形式的指令编码较短。

2.4.2 单项运算指令

这种指令的形式如下：

X←○X　　○：算符

不能采用Y←○X的形式。

单项运算指令包括增量、减量、符号反转和位反转指令。符号反转指令就是将带符号的二进制数变成绝对值相等、符号相反的数据的指令。该指令实际运算时，是从$FFFF_{16}$减去原数再加1。

单项运算指令有下面两种形式，这里以助记符INC为例说明。

（1）INC r/m

这种形式是对寄存器/存储器运算的指令。在对存储器运算时，先读出存储器的内容，然后再写入运算结果。这时标志要发生变化。

（2）INC reg

只在INC、DEC指令中才有这种形式。在（1）中也有与此相同功能的指令，但这种形式的编码较短。

2.4.3 十进制校正指令

这是为十进制数据而准备的AL、AX寄存器校正指令。这种指令可以对压缩型十进制数加减运算的结果进行校正，也可以对非压缩型十进制数的加减乘运算结果及除法运算的被除数进行校正。DAA、DAS中的D表示十进制数，AAA、AAS、AAM、AAB中前头的A表示ASCII码。在非压缩型十进制数的高4位上加3就变成了ASCII码的数据（参见图2.3及图2.4）。

（1）DAA

DAA指令是压缩型十进制数的加法校正指令。如果辅助进位标志为1，或者AL寄存器的低4位的内容在A_{16}以上时，则AL寄存器的内容加6，并设置辅助进位标志。如果其结果的进位标志为1，或者AL寄存器的内容为$A0_{16}$以上时，再在AL寄存器的内容上加60_{16}，并设置进位标志。

（2）DAS

DAS指令是压缩型十进制数的减法校正指令。如果辅助进位标志为1，或者AL寄存器的低4位的内容在A_{16}以上时，则从AL寄存器的内容中减6，并设置辅助进位标志。如果其结果的进位标志为1，或者AL寄存器的内容在$A0_{16}$以上时，再从AL寄存器的内容减去60_{16}，并设置进位标志。

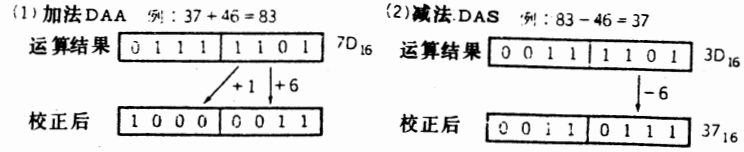
（3）AAA

AAA指令是非压缩型十进制数的加法校正指令。如果辅助进位标志为1，或者AL寄存器的低4位的内容在A_{16}以上时，就在AX寄存器的内容中加上106_{16}，并设置辅助进位和进位标志。并且，将AL寄存器高4位的内容置0。

（4）AAS

AAS指令是非压缩型十进制的减法校正指令。如果辅助进位标志为1，或者AL寄存器中低4位的内容在A_{16}以上时，就从AL寄存器的内容中减去6，并设置辅助进位和进位标志。同时再从AH寄存器的内容中减1，将AL寄存器高4位的内容置为0。

1. 压缩型10进制校正



2. 非压缩型10进制的校正

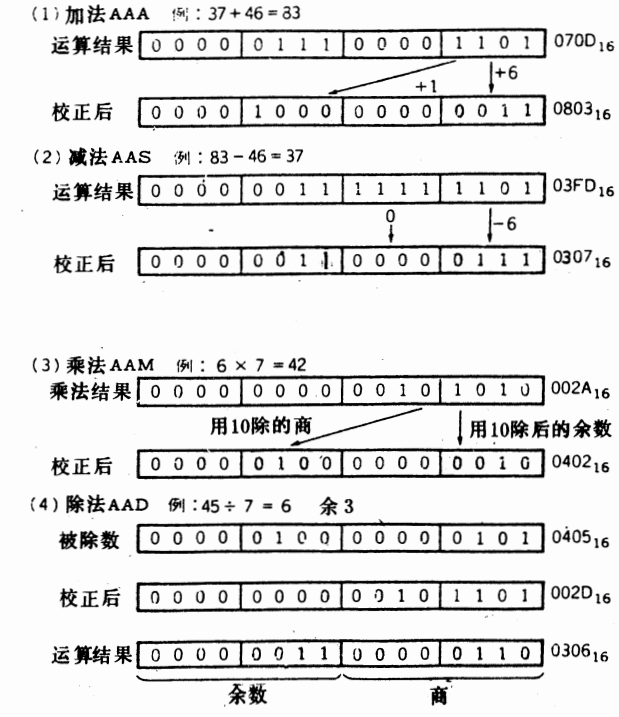


图2.4 用校正指令进行AL、AX寄存器的校正 1. 压缩型十进制校正 2. 非压缩型十进制校正

（5）AAM

AAM指令，从助记符上看是非压缩型十进制数的乘法校正，但实际上是0_{10}～99_{10}的二进制数向非压缩型十进制数的变换指令。在乘法运算之后，采用该指令可以进行非压缩型十进制数的校正，因此而得名。AL寄存器的内容除以10，所得的商存入AH寄存器，余数存入AL寄存器。

（6）AAD

AAD指令，从助记符上看是非压縮型十进制数的除法校正，但实际上是0_{16}～99_{16}的非压縮型十进制向二进制的交换指令。由于除法运算在执行之后进行修正不能得到正确的商。为此，事先用该指令将被除数变换成二进制数，然后再进行除法运算。最后再通过AAM指令把商变回到十进制数。AH寄存器的内容10倍之后加到AL寄存器中，AH寄存器的内容变为0。

2.4.4 乘除运算指令与数据扩充指令

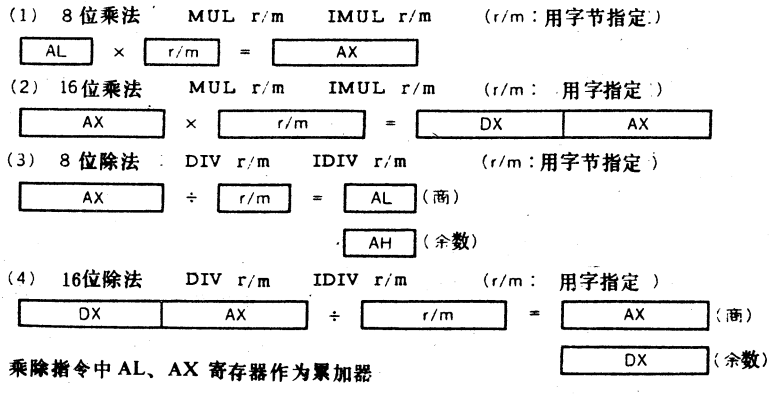


图2.5 乘除运算指令的类型

8086微处理器可以进行无符号及带符号数据的乘除法运算。在乘除运算指令中，作为累加器的寄存器是固定的，可采用AL、AX、DX寄存器。乘数、除数可以从寄存器/存储器中选出。乘除运算指令的类型如图2.5所示。在汇编语言上是进行8位运算还是进行16位运算，是通过寄存器/存储器以字节定义，或者以字定义来指定的。在乘除运算中，由于带符号数据与无符号数据的算法不同，所以分别采用不同的指令。

在除法指令中，被除数的位数是除数位数的二倍。因此，8位数据除以8位数据，或者16位数据除以16位数据时，必须对数据进行扩充。对无符号数据的除法运算来说，将0分别装入AH、DX寄存器中。而在带符号数据的除法运算中，则应该将符号位的内容装入AH、DX寄存器里。CBW、CWD指令就是为此而准备的，它可以把AL、AX寄存器最高位的内容装入AH、DX寄存器中。

（1）无符号乘法指令MUL r/m

这是无符号的二进制数乘法指令。下面为乘法指令的例子。例中表示8位数乘8位数据。

```
BDAT0  DB     48
BDAT1  DB     57
       ⋮
       MOV    AL, BDAT0
       MUL    BDAT1
```

（乘积存入AX寄存器中）

（2）带符号乘法指令IMUL r/m

这是带符号的二进制数乘法指令。其程序举例如下所示，例中为16位数据乘16位数据。

```
WDAT0  DW     3287
WDAT1  DW     -4562
       ⋮
       MOV    AX, WDAT0
       IMUL   WDAT1
```

（乘积存入DX、AX寄存器中）

（3）无符号除法指令DIV r/m

这是无符号的二进制数除法运算指令。如果除法运算结果溢出时（比如在8位除法运算时执行512÷2），就会无条件产生类型0的中断（参考1.4）。其程序举例如下：

a）16位数据÷8位数据

```
WDATO  DW   3287
BDATO  DB   47
       ⋮
       MOV  AX, WDATO
       DIV  BDATO
```

（商存入AL寄存器，余数存入AH寄存器）

b）8位数据÷8位数据

```
BDAT0  DB   236
BDAT1  DB   23
       ⋮
       MOV  AL, BDAT0
       MOV  AH, 0
       DIV  BDAT1
```

（商存入AL寄存器，余数存入AH寄存器）

（4）带符号的除法指令IDIV r/m

这是带符号二进制数的除法运算指令。如果除法运算结果溢出时，就无条件产生类型0的中断。其程序举例如下：

a）32位数据÷16位数据

```
DDATL  DW   1362H
DDATH  DW   4953H
WDATO  DW   -27349
       ⋮
       MOV  AX, DDATL
       MOV  DX, DDATH
```

```
        DIV   WDATO
```

（商存入AX寄存器，余数存入DX寄存器）

b）16位数据÷16位数据

```
WDAT0  DW    31574
WDAT1  DW    -287
       ⋮
       MOV   AX, WDAT0
       CWD
       DIV   WDAT1
```

（商存入AX寄存器，余数存入DX寄存器）

在上述各例中，为了简便起见，将变量的定义和指令一起描述，但通常情况下，变量在数据段中定义。

带符号数据的除法运算中，对负数的除法来说，由于余数符号的取法不同，可以有下面两个不同的解。在8086微处理器中，余数的符号与被除数的符号一致，如表2.4所示。

$-7 \div 2 = -3$　余数为 -1

$-7 \div 2 = -4$　余数为 $+1$

表2.4 商与余数的符号

被除数	除数	商	余数
正	正	正	正
正	负	负	正
负	正	负	负
负	负	正	负

8086微处理器中，余数的符号与被除数的符号一致。

（5）CBW

CBW（变换字节为字的指令）指令将AL寄存器中的8位二进制数变换成16位，装入AX寄存器中。在进行带符号的8位数据除法运算（IDIV）时，如果被除数为8位，则首先采用CBW指令将

8位扩充到16位，然后再运算。CBW，CWD指令用于在除运算之前扩充带符号的数据。参阅图2.6。

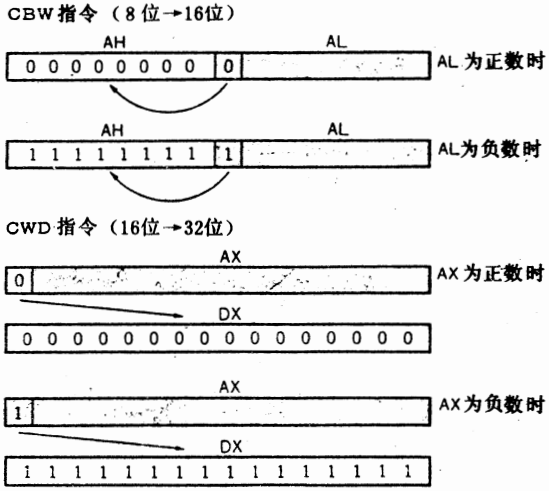


图2.6 CBW、CWD指令的功能

（6）CWD

CWD（变换字为双字指令）指令将AX寄存器中带符号的16位二进制数变换成32位，装入DX、AX寄存器中。在进行带符号的16位数据的除法运算（IDIV）时，如果被除数为16位，则采用CWD指令首先将16位扩充到32位，然后再运算（参见图2.6）。

2.4.5 2的补码表示法及带符号的运算

（1）2的补码表示方法

8086微处理器可以进行带符号数据的运算，为了很好地适用这

些指令，必须理解带符号数据的表示方法。

表2.5列出了4位的二进制数与带符号的十进制数的对应情况。通常，带符号的二进制数的最高位为符号位，并定义该位为0时表示正，1时表示负。带符号的二进制数，由于负数的表示方法不同，有下述三种情况：

a）绝对值表示

这种方法就是将我们日常所使用的十进制数原封不变地变换为二进制数，最高位为符号位，其余的位表示绝对值。这种表示方法对人来说很容易理解，但在计算机中如使用这种方法，会使运算器变得很复杂，所以不能采用。

b）1的补码表示

表2.5 二进制数的负数表示方法

二进制码	无符号二进制数	带符号二进制数		
		绝对值	1的补码	2的补码
0000	0	+0	+0	0
0001	1	+1	+1	+1
0010	2	+2	+2	+2
0011	3	+3	+3	+3
0100	4	+4	+4	+4
0101	5	+5	+5	+5
0110	6	+6	+6	+6
0111	7	+7	+7	+7
1000	8	-0	-7	-8
1001	9	-1	-6	-7
1010	10	-2	-5	-6
1011	11	-3	-4	-5
1100	12	-4	-3	-4
1101	13	-5	-2	-3
1110	14	-6	-1	-2
1111	15	-7	-0	-1

这种表示方法是将负数取其绝对值相等的正数的反码（1的补码）来表示。如将绝对值相等的正数与负数相加，其结果等于11…1。这种表示方法的优点是包括进位、借位的正负数加减运算可以采用统一的运算器，但由于0有两种表示方法，所以最近一般不使用。

c）2的补码表示

这种表示方法是将负数取绝对值相等的正数的各位取反，然后再加1来表示。如将绝对值相等的正负数相加，其结果为0 0 …0与进位。这种方法运算器略为复杂一些，但由于0具有单值性，加减法运算与无符号二进制数的计算方法相同，所以，8086和其他很

表2.6 8086中能处理的带符号数据

8位数据		16位数据	
二进制	十进制	二进制	十进制
01111111	+127	0111111111111111	+32767
01111110	+126	0111111111111110	+32766
01111101	+125	0111111111111101	+32765
⋮		⋮	
00000001	+1	0000000000000001	+1
00000000	0	0000000000000000	0
11111111	-1	1111111111111111	-1
11111110	-2	1111111111111110	-2
⋮		⋮	
10000010	-126	1000000000000010	-32766
10000001	-127	1000000000000001	-32767
10000000	-128	1000000000000000	-32768

多计算机中都采用这种方法。

表2.6列出了在8086微处理器中能处理的带符号数据的范围。

（2）用2的补码表示的二进制运算

下例为2的补码表示的二进制数运算。为了简便起见，例中是用4位二进制数表示的，不过8086对8位及16位的二进制数都可同样进行运算。例中左侧的计算为8086实际执行的二进制数据。SI、US下面所表示的数值分别为带符号、无符号的二进制数据用16进制表示的值。

（1）加法

6 + (-2) = 4

二进制	SI	US
0110	6	6
+ 1110	-2	E
10100	4	4 + CR

8 + (-7) = -4

二进制	SI	US
0011	8	8
+ 1001	-7	9
1100	-4	C

（2）减法

4 -(-8)= 7

二进制	SI	US
0100	4	4
- 1101	-8	D
10111	7	7 - BR

8 - 7 = -4

二进制	SI	US
0011	8	8
- 0111	7	7
11100	-4	C - BR

（8）乘法

8 × (-2) = -6

二进制	SI	US
0011	8	8
× 1110	-2	E
11111010	-6	FA

（IMUL指令的结果）

二进制	SI	US
0011	3	3
×1110	-2	E
00101010	2A	2A

（MUL指令的结果）

SI：带符号数据　　CR：进位

US：无符号数据　　BR：借位

从例中可以看出，对加减法来说，把二进制数无论看成带符号的数，还是看成无符号的数，都可得到正确的结果。但在乘法运算中，要想带符号的数据得到正确的结果，无符号的数据就得不到正确结果；反之，要想无符号的数据得到正确结果，带符号的数据就得不到正确结果。除法运算也存在同样情况。为此，在8086微处理器中，为了使两者都能得到正确的结果，对带符号数据和无符号数据准备了两种乘除运算指令。

（3）溢出标志

在上例中，带符号数据的值尽管没有考虑到进位、借位的内容，仍然可以得到正确的结果。这是因为在带符号的数据中，最高位是符号位，所以进位、借位的值没什么意义。在带符号数据的运算中，溢出问题是很重要的。所谓溢出就是指加减运算的结果超出了所能表示的范围。比如，8086在进行下述运算时，就要设置溢出标志。

a）8位运算

$98+49=147$（>127）

$-82-53=-135$（<-128）

b）16位运算

$29825+18637=48462$（>32767）

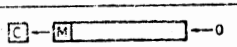
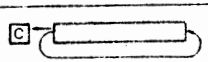
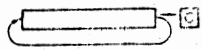
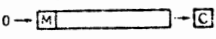
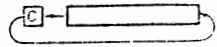
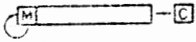
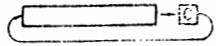
$-18321-28465=-46786$（<-32768）

通常，在产生溢出时，以后的处理就不能得到正确的结果。因此，在执行运算指令之后立即执行INT0指令，如果产生中断就转移到中断处理程序。在乘法指令中不会产生溢出。在除法指令中如果产生溢出，就自动地转移到类型0的中断处理程序。

2.5 移位/循环指令

8086微处理器中有表2.7所示的8种移位/循环指令。移位或循环的对象是寄存器/存储器。

表2.7 移位/循环指令

移位指令			循环指令		
助记符	意义	内 容	助记符	意义	内 容
SHL	逻辑左移		ROL	左旋	
SAL	算术左移	同 上 (注)	ROR	右旋	
SHR	逻辑右移		RCL	带进位左循环	
SAR	算术右移		RCR	带进位右循环	

C：进位标志　　M：最高位（符号位）

（注）SHL与SAL是相同的指令，只是助记符不同。

算术移位指令是符号位（最高位）不变，其余各位移位的指令。由于算术左移位与辑逻左移位的结果相同，所以尽管助记符是两种，但指令编码却只有一个。

移位/循环指令有两种类型，一种指令是移位数、循环数只有1位的指令，另一种指令是由CL寄存器指定移位数、循环数的指令。下面是SHL指令的例子，其他指令也与SHL指令的描述完全相同。

(1) SHL r/m, 1

〔例〕SHL AX, 1

SHL BDATO, 1

(2) SHL r/m, CL

〔例〕MOV CL, 4

SHL AX, CL

SHL BDATO, CL

在移位/循环指令中，有把CL寄存器作为计数器使用的指令，在执行这条指令后，CL寄存器的内容不变。

2.6 分支转移指令

8086微处理器的分支转移指令包括无条件转移、调用、返回、条件转移和中断指令，8086中沒有8085A中的条件调用和条件返回指令。转移目标地址除了向段外分支转移和间接地址之外，都是指令指针相对地址（参考1.3.3）。指令指针相对的转移地址要以（转移、调用指令的地址）+ 2作为基址，这一点务必注意。

2.6.1 无条件转移、调用指令

无条件转移指令和调用指令，除了下述两点外，二者都有相同类型的指令。

- 沒有短调用（Short Call）指令。
- 调用指令是在指令指针的内容(段外调用时还包括CS寄存器的内容）进栈后再转移。

（1）JMP label……短相对

以当前的地址单元 + 2 为基址，转移到－128～＋127的地址。如果转移的目的地址很近时，指令就可以短些。在调用指令中沒有这种类型的指令。

（2）JMP label; CALL label

……段内相对

这种指令是向代码段内的任意地址转移。对调用指令来说，只有指令指针进入堆栈。（虽然位移量是－32768～＋32767，但是超出65534的地址要向 0 地址转移，所以可以向段内所有的地址转移）。

（3）JMPF label; CALL label

……段外直接

这是向其他段转移的指令。地址由操作码后面的四个字节（偏移和段）指定。对于调用指令来说，先将CS寄存器、指令指针的内容顺序进入堆栈后再转移。

（4）JMP r/m; CALL r/m

……段内间接

这指令以寄存器/存储器指定的寄存器或存储器的内容（2字节）为偏移地址进行转移。例如：

```
        JMP     AX
        CALL    POINT
POINT   DW      OFFSET  TRGT
```

（这里TRGT为转移目的地址的标号）

（5）JMPF label, CALLF label

……段外存储器间接

以寄存器/存储器所指定的存储器内容的低端2字节为位移地址，高端2字节为段地址进行分支转移。寄存器/存储器在指定寄存器时，工作不定。对于调用指令来说，先将CS寄存器、指令指针的内容送入堆栈后再转移。例如：

```
        JMPF   POINT
POINT   DD     TRGT
```

（这里的TRGT为转移目的地址的标号）

2.6.2 条件转移指令

表2.8 条件转移指令及转移指令

助记符	意　义(注)	条　件	
JE JZ	“等于”和“全零”转移	Z = 1	(A)=(B)
JNE JNZ	“不等于”和“不为零”转移	Z = 0	(A)≠(B)
JB JNAE	“低于”和“不高于/不等于”转移	C = 1	(A)<(B)无符号
JNB JAE	“不低于”和“高于/等于”转移	C = 0	(A)≥(B)无符号

助记符	意　义(注)	条　件	
JA JNBE	“高于”和“不低于/不等于”转移	C∨Z＝0	(A)＞(B)无符号
JNA JBE	“不高于”和“低于/等于”转移	C∨Z＝1	(A)≦(B)无符号
JL JNGE	”小于”和“不大于/不等于”转移	S∨O＝1	(A)＜(B)带符号
JNL JGE	“不小于”和“大于/等于”转移	S∨O＝0	(A)≧(B)带符号
JG JNLE	“大于”和“不小于/不等于”转移	(S∀O)∨Z＝0	(A)＞(B)带符号
JNG JLE	“不大于”和“小于/等于”转移	(S∀O)∨Z＝0	(A)≦(B)带符号
JS	符号标志置1转移	S＝1	上面的条件是表示前面执行CMP　A，B或SUB　A，B指令时比较A、B大小的条件。（A、B为寄存器或存储器）
JNS	符号标志置0转移	S＝0	
JO	溢出转移	O＝1	
JNO	无溢出转移	O＝0	
JP JPE	“有奇偶性”和“奇偶性为偶”转移	P＝1	
JPO JNP	“奇偶性为奇”和“无奇偶性”转移	P＝0	
JCXZ	CX＝0转移指令	CX＝0	

Z：零标志的内容　　C：进位标志的内容
S：符号标志的内容　　O：溢出标志的内容
P：奇偶校验标志的内容　　CX：CX寄存器的内容
∨：“或”　　∀：“异　或”
注：指令意义请参考附录1。

该指令用于判别标志或CX寄存器的内容，如果条件满足就转移。转移地址只有短类型，即在以当前地址单元＋2为基址的－128～＋127的地址范围内。表2.8列出条件转移指令。表中上面的10个指令，每个指令编码都可以使用两个助记符。

条件转移指令是进行大小比较及判定CX寄存器内容的指令，这条指令是8085A中所没有的。大小比较指令的助记符不是表示标志的内容，而是表示大小比较的条件，所以只看助记符不太容易理解，而看表中最右一栏将会更容易理解。助记符具有下述的意义：

Below　小于无符号数据

Above　大于无符号数据

Less　小于带符号数据

Greater　大于带符号数据

2.6.3 中断指令

这是由软件产生中断的指令，相当于8085A的RST指令。中断时分支转移的过程请参考1.4。对中断的分支转移来说，CS寄存器、指令指针前面的标志位内容也要进入堆栈，这一点应该注意。

（1）INT n（n＝0～255）

该指令产生类型n的中断。

（2）INT3

这是1字节指令，在（1）指令中，如果n＝3与本指令的功能相同。通过软件设定断点时使用这种指令。

（3）INTO

这条指令表示只在溢出标志为“1”时才产生中断。它用于判定带符号数据的加减运算的结果是否产生了溢出。

2.6.4 返回指令

返回指令中包括段内调用的返回、段外调用的返回和中断的返回等三种指令。段内调用返回指令和段外调用返回指令在返回之后，堆栈指针的内容中可以加立即数。这条指令是为了在子程序调

用时，对数据部分的堆栈指针进行更新。（由于进栈、出栈是以字为单位进行的，所以立即数据通常为偶数。）

（1）RET〔n〕……段内

该指令表示，只是指令指针的内容出栈。如果有n(0～65536)，则在出栈后，堆栈指针中加n。

（2）RETF〔n〕……段外

该指令表示，按指令指针、CS寄存器的顺序出栈。如果有n，则出栈后，在堆栈指针中加n。

（3）IRET

该指令表示，按指令指针、CS寄存器、标志的顺序出栈。该指令用于中断返回。

2.7 重复指令

重复指令就是把CX寄存器作为计数器，重复执行相同的指令。重复指令可以分为字符串处理指令和循环指令两大类。

2.7.1 字符串指令

这种指令就是对存储器以块（字符串）为单位进行处理的指令、它包括传送、存储、比较和扫描指令（参考图2.7）。字符串处理指令必须与重复的前缀一起使用，通过选择不同的重复前缀，在比较、扫描指令中，无论比较的结果一致还是不一致，都可以使处理停止。

在字符串处理指令中，下述的寄存器、标志可作存储器的指针、计数器等使用。因此这些寄存器、标志必须事前设置好。程序例5的（A）表示块传送的例子。

a）SI寄存器、DS寄存器

这些寄存器用在传送、比较指令中指定源地址。在处理完1个字节（或1个字）之后，SI寄存器的内容加1（或2）或者减1（或2），指令执行结束后，保持该值不变。在执行字符串处理指令过

程序例5　重复指令的举例

```
  1000                DATSEG  EQU   1000H
  2000                EXTSEG  EQU   2000H
  00C8                NARRAY  EQU   200
                      ;
                              CSEG
                              ORG   0
                      ; (A) ------------------------------------
0000 B8 00 10                 MOV   AX,DATSEG
0003 8E D8                    MOV   DS,AX
0005 B8 00 20                 MOV   AX,EXTSEG
0008 8E C0                    MOV   ES,AX
                      ;
                      ; (B) ------------------------------------
                      ;       模块传送
000A BE 00 00                 MOV   SI,OFFSET DARRAY 源指针
000D BF 00 00                 MOV   DI,OFFSET EARRAY 目标指针
0010 B9 64 00                 MOV   CX,NARRAY/2      计数器
0013 FC                       CLD
0014 F3 A5                REP MOVS  AX,AX ←——— 指定传送以字为单位
                      ;
                      ; (C) ------------------------------------
                      .       软件计时器
0016 B9 64 00                 MOV   CX,100
0019 90               CONT:   NOP
001A 90                       NOP
001B E2 FC                    LOOP  CONT
                      ;
                      ; (D) ------------------------------------
001D B9 C8 00                 MOV   CX,NARRAY
0020 BB 00 00                 MOV   BX,OFFSET DARRAY
0023 8A 07            CHK0:   MOV   AL,[BX]
0025 84 C0                    INC   BX
0027 43                       TEST  AL,AL ←—— 如果在DARRAY中有D以
0028 E1 F9    0023            LOOPZ CHK0      外的数就转出循环
                      ;
                      ; (E) ------------------------------------
002A B9 64 00                 MOV   CX,100 该例中由于循环内CX寄存器赋值
002D 90               OVR:    NOP          为2，所以不能从循环中转出
002E B9 02 00                 MOV   CX,2
0031 E2 FA                    LOOP  OVR
                      ;
                      ; (F) ------------------------------------
                      ;
  1000                        DSEG  DATSEG
0000                  DARRAY  RS    NARRAY
                      ;
  2000                        ESEG  EXTSEG
0000                  EARRAY  RS    NARRAY
```

程中，每处理1个字节（或字）就要检查是否有中断请求，如果有就产生中断。但是在中断结束后，只有字符串指令之前的前缀才是有效的，其他的前缀都变为无效。因此，在执行带有段更换前缀、锁定前缀的字符串指令时，或禁止中断，或事先编写一段下面的程序。

```
CONTREP:
 LOCK  REP     MOVS    AX, CS  ;  AX
       TEST    CX,  CX
       JNZ     CONTREP
```

通过段更换，源地址也可使用CS、SS、ES寄存器。其中：

段　DS寄存器

偏移　SI寄存器

b）DI寄存器、ES寄存器

这两个寄存器在所有的指令中，都用于指定目标地址。（在比较、扫描指令中，该寄存器指定的地址作为减数。）处理1字节（或1字）后，DI寄存器加1（或2）或者减1（或2），指令执行结束后，保持该值不变。其中：段　ES寄存器

偏移　DI寄存器

c）CX寄存器

该寄存器用于指定块的长度（即处理的计数值）。每处理完1字节（或字）都要减1（或2），直到等于0时，处理结束。

d）方向标志

该标志表示处理的方向。在方向标志为“0”时，DI、SI寄存器每处理1个字节（或字）就相加，反之，当方向标志为“1”时就相减。方向标志可通过STD、CLD指令直接设置和复位。

（1）重复前缀

重复前缀包括下面三种指令，这三种前缀都必须加在字符串指令之前。

a）REP

REP加在传送、存储指令之前。在CX寄存器的内容等于0之前，不断进行重复处理。

b) REPZ/REPE

REPZ/REPE（重复或迭代前缀）加在比较、扫描指令之前。运算的结果的零标志为“1”时，即使CX寄存器的内容不为0，也要停止处理。这时CX、SI、DI寄存器要保持当时的数值不变。虽然有两种不同的助记符，但指令编码相同。字符串处理停止的判定是在各寄存器的内容更新之后进行。因此，在进行最初的字节(或字)处理时，处理如果停止，CX、DI、SI寄存器的内容进行一次更新。

c) REPNZ/REPNE

该重复前缀与REPZ/REPE相比，停止处理的条件是零标志为“0”，而不是“1”。除这一点外，二者完全相同。

（2）MOVS AX，AX（AL，AL）

MOVS指令是将存储器中的某一块内容传送到另外一块去的指令，可以字节为单位处理，也可以字为单位处理。（存储、比较、扫描指令也一样。）

（3）STOS AX(AL)

STOS指令是将AL或AX寄存器的内容传送到附加段中DI所指向的单元的指令，通常用于在存储器中预置某个数值。

（4）CMPS AX，AX（AL，AL）

CMPS指令表示从SI寄存器所指定的存储器内容，减去DI寄存器所指定的存储器内容。该指令只影响标志，不改其他内容。通常用于存储器块间的比较，通过重复前缀的选择不同，一致、不一致都可以判断。

（5）SCAS AX（AL）

SCAS指令表示从AL或AX寄存器的内容减去DI寄存器指定的存储器内容。只影响标志，而其他内容不变。通过重复前缀的选择不同，块内的相同数据、不同数据都可以判断。

在CX寄存器的内容为0时，不论执行那条字符串指令都相当于无任何操作，直接转到下一条指令。

2.7.2 循环指令

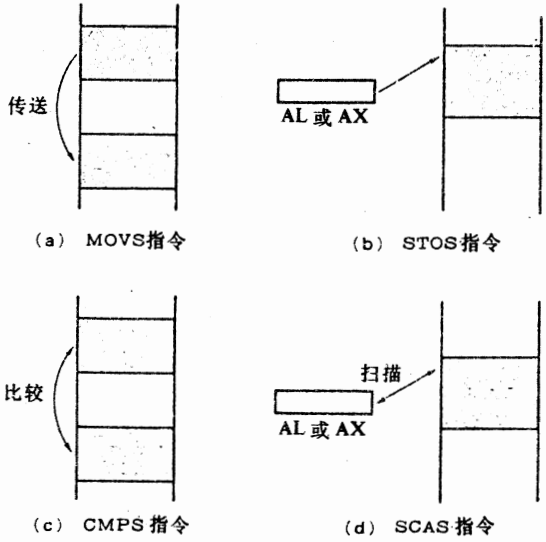
循环指令是把CX寄存器作为计数器，重复执行相同程序的指令。因此，在循环的环路内不能有改变CX寄存器内容的指令。循环指令能够转移的范围以循环指令所在的地址单元+2为基址，−128～+127的范围内。循环结束时，CX寄存器要保持当时的数值。程序例5就是使用循环指令的例子。

（1）LOOP label

该指令表示CX寄存器的内容减1后，如不为0就转移到标号所指的地址单元。（CX寄存器的内容减1，不会引起标志的变化。）

（2）LOOPZ/LOOPE label

该指令表示CX寄存器的内容减1后，如不为0，且零标志为



（a） MOVS指令　（b） STOS指令

（c） CMPS指令　（d） SCAS指令

图2.7 字符串处理指令

“1”时转移。助记符有两种，但指令编码相同。

（3）LOOPNZ/LOOPNE label

除了转移条件是零标志为“0”外，其他都与LOOPZ/LOOPE指令相同。

在上述循环指令的条件判定是在CX寄存器减1之后进行的。因此，一次也没进行循环时，CX寄存器的内容比原来的值小1。

如果CX寄存器的内容为0时执行循环指令，则CX寄存器的内容就成了$FFFF_{16}$，并进行转移，要循环65536次。

2.8 标志操作指令

标志操作指令是对标志直接置位、复位的指令。只有进位标志、方向标志、中断允许标志可以直接由软件控制。

（1）STC、CLC、CMC

STC、CLC、CMC分别表示对进位标志进行置位、复位、求反的指令，而不影响其他的标志。

（2）STD、CLD

STD、CLD分别表示对方向标志置位、复位的指令，详细请参阅2.7.1及附录1。

（3）STI、CLI

STI、CLI分别表示对中断允许标志置位、复位的指令。相当于8085A中的EI、DI指令。

2.9 其他指令

（1）NOP

NOP指令为空操作指令。其指令编码（90_{16}）与XCHG AX，AX指令相同。指令编码不是00_{16}，这一点应该注意。

（2）HLT

HLT指令执行时，8086微处理器进入暂停状态。要想解除暂停状态重新启动，必须从外部产生中断請求。

（3）LOCK

LOCK指令是总线锁定前缀。在执行带有LOCK前缀的指令期间，8086的$\overline{\text{LOCK}}$端处于低电平。详细請参考1.5.3。

（4）WAIT

WAIT指令是为了与其他的处理器（通常是辅助处理器8087）同步的信号。执行WAIT指令时，8086处于等待状态，直到TEST端上有低电平的信号输入为止。在等待状态仍然可以检出中断請求，一旦有外部中断請求，就产生中断，中断结束后再进入等待状态。

等待指令通常用于等待8087的运算结果。8086的TEST端与8087的BUSY端（运算结束时为低电平）相连接。由于8086和8087是相互并行执行指令，所以当8086需要8087的运算结果时，就必须等待8087的运算结束，WAIT指令就是为此而准备的。

（5）ESC

ESC（处理器交权指令）指令是辅助处理器8087的指令。8086与8087是并行执行同一个程序。如果在程序中有8086的指令，则8086就执行该指令，8087就不执行。反之，执行的指令是ESC指令（在8087中以另外的助记符定义），则8087执行该指令。这时，如果该指令是存储器存取指令，则8086计算出有效地址，执行空（dummy）的读周期。（除此之外的ESC指令，8086都不执行。）如果该指令是存储器读指令，则8087在空的读周期读数据总线上的数据。如果是存储器写指令，则在空的读周期中把该地址锁存，然后在该地址上执行写的操作。

3 外围芯片

在实际使用8086微处理器时，还需要配有时钟发生器、总线控制器等若干外围芯片。本章将说明这些芯片的概要及使用上的注意事项。

3.1 时钟发生器8284A

8086微处理器芯片内部没有晶体振荡器，因此必须从外部提供时钟。8284A 就是提供时钟的芯片，芯片中包括时钟产生电路、复位信号产生电路和就绪信号控制电路，分别向8086提供时钟、复位和就绪信号。图3.1、图3.2分别表示8284A 芯片的引脚排列图和内部逻辑电路图。

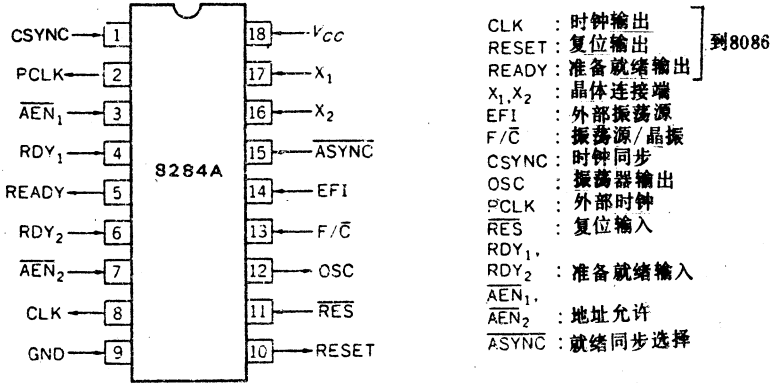


图3.1 8284A的引脚排列图

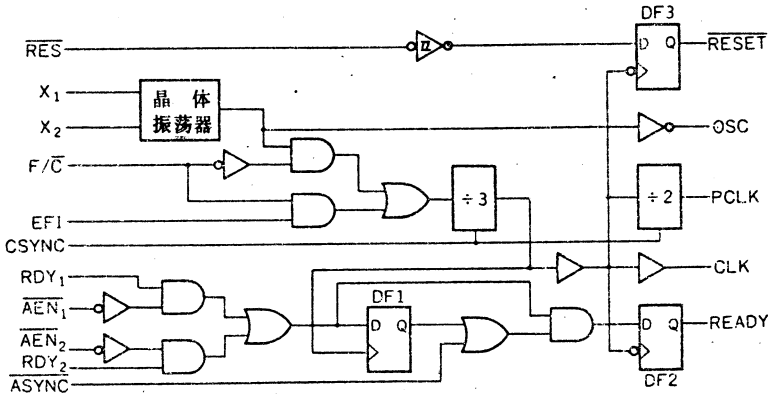


图3.2 8284A的内部电路

（1）复位信号产生电路

复位信号产生电路采用斯密特触发输入的复位脉冲发生器。将输入的信号与时钟发生电路产生的时钟相同步后再输出。由于$\overline{RES}$输入是斯密特触发输入，因此可以直接连接DF3。

（2）时钟发生电路

时钟发生电路由晶体振荡器和分频器组成。在晶体振荡器上产生的信号，首先通过1/3分频器分频成脉冲宽度L与H之比为1:2的时钟信号，CLK输出给8086。这时F/$\overline{C}$端为低电平。如果将F/$\overline{C}$端连接成高电平，也可以使用从EFI端输入的外部信号作为基本时钟。OSC、PCLK可作为外围大规模集成电路（LSI）用的通用时钟输出。OSC信号是反相之后输出，这一点必须注意。

CSYNC输入信号是为了与其他的8284A芯片时钟同步的信号。当CSYNC为高电平时，CLK、PCLK信号被强制为高电平（同步预置）。3分频、2分频计数器从CSYNC为低电平后的下一个分频时钟的上升沿开始计数。CSYNC至少在2个分频时钟周期中保持高电平。CSYNC信号必须和分频时钟同步输入，如果不同步，就通过图3.3的电路实现同步。

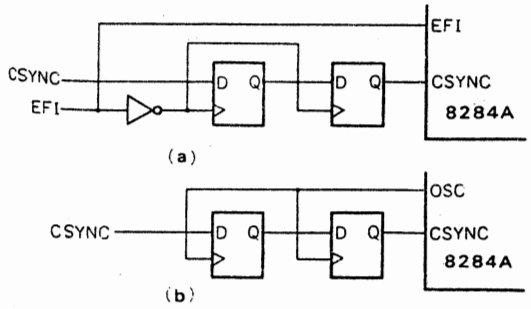


图3.3 CSYNC的同步电路
(a)外部时钟 (b) 内部时钟

(3) 就绪控制电路

就绪控制电路由AND、OR门电路和触发器组成，就绪信号与时钟的下降沿同步后输出。该电路以常态非就绪型 (Normal not ready type) 为前提工作，不能用在常态就绪型 (Normal ready type)，这是因为READY信号是与时钟的下降沿同步输出造成的。(参阅4.1)。

从 $\overline{AEN_1}$、$\overline{AEN_2}$ 端输入外围芯片的选择信号，RDY_1、RDY_2 端在一定的等待周期后输入高电平。

$\overline{ASYNC}$ 端是进行一级同步，还是进行二级同步的选择输入端。当RDY_1、RDY_2与时钟同步，并满足时钟的建立时间时，采用一级同步即可，这时$\overline{ASYNC}$端为高电平或开路（即$\overline{ASYNC}$悬空)，参阅图3.4(a)。而在RDY_1、RDY_2与时钟异步输入，不满足建立时间时，就需要二级同步，$\overline{ASYNC}$为低电平。这时RDY_1、RDY_2的取样波形要提前1/3个时钟。参阅图3.4(b)。

图3.4中的尖脉冲是由于触发器内部门电路延迟时间的偏移而产生的。这种尖脉冲不只是在8284A上会产生，在所有的触发器中都可能产生，因此在硬件设计时要特别注意。

如果在时钟频率为5MHz情况下使用8086时，该尖脉冲在RE-

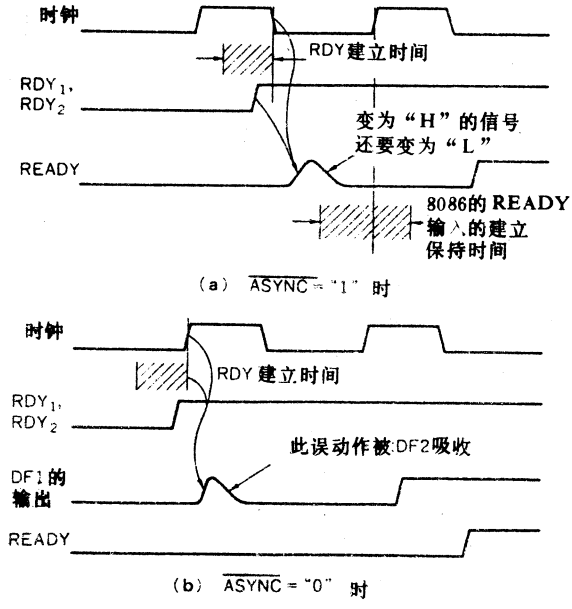


图3.4 $\overline{\text{ASYNC}}$端与READY输出的尖脉冲

ADY信号建立之前已经消失，就不会产生什么问题。

3.2 总线控制器8288

总线控制器8288芯片只有最大方式才需要。在最大方式中，命令信号和总线控制所需要的信号都是从8288输出的。8288利用8086、8087、8089输出的$\overline{S_0}$～$\overline{S_2}$三个状态信号和时钟信号产生所需的信号。

图3.5、图3.6表示8288的引脚排列图和方框图。

8288芯片中的状态译码器对从8086输出的状态进行译码，产生内部所需要的信号。命令信号产生电路和控制信号产生电路再利用

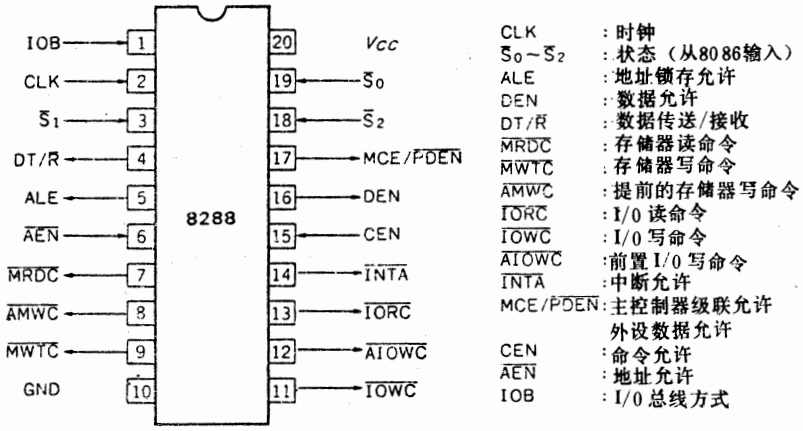


图3.5　8288的引脚排列图

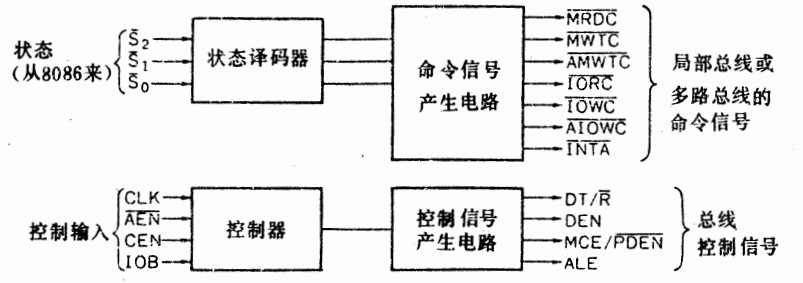


图3.6　8288的方框图

上述信号产生命令信号和总线控制信号（关于命令信号和总线控制信号請参阅4.1）。

CEN端是使用2个以上8288时采用的信号，当CEN信号为低电平时，命令信号、DEN、$\overline{PDEN}$信号被强制变成无效信号。在通常情况下，CEN端为高电平。在使用2个以上8288时，只有正在控制存取的那个8288的CEN端为高电平。

$\overline{AEN}$端是支持多总线结构的引线端，在命令信号输出给多总线时，要与8289的$\overline{AEN}$输出端相连接。在$\overline{AEN}$端为高电平的期

间内，命令信号为浮动状态。当获得总线，$\overline{AEN}$变为低时，命令信号就变成高，并在115～200nS（毫微秒）之后，$\overline{MRDC}$、$\overline{MWTC}$、$\overline{IORC}$、$\overline{IOWC}$之一将变为低电平。这些信号都是为了满足多总线同步条件而准备的，如图3.7所示。

IOB端的信号，当只把8288用于控制I/O芯片时为高电平。在通常情况下将IOB端置于低电平，用作对存储器和I/O两方面的控

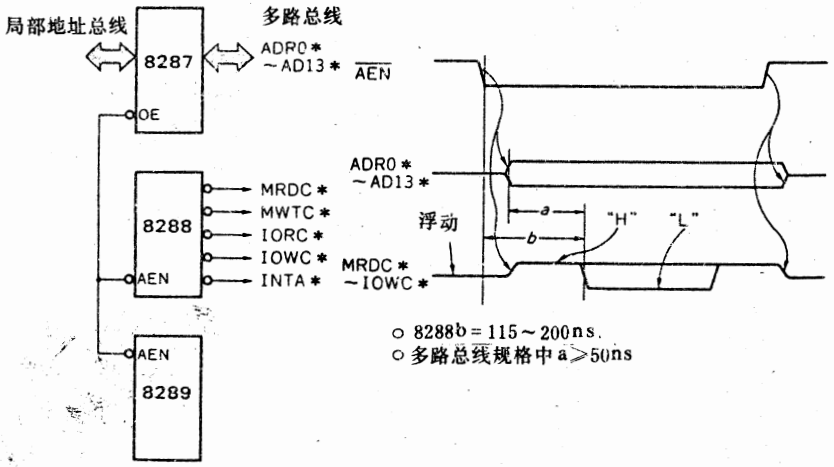


图3.7　8288的$\overline{AEN}$与命令信号输出

制。如果IOB端信号为高电平，只在访问I/O时才会使$\overline{IORC}$、$\overline{IOWC}$、$\overline{AIORC}$、$\overline{INTA}$信号有效，而在存取存储器时则不进行任何操作。在这种方式时，由MCE/PDEN端输出PDEN信号。该信号是与DEN信号时序相同，而逻辑相反的信号，这些信号连接在总线缓冲器上。在这种方式时，不必考虑$\overline{AEN}$端信号的状态。

在IOB端为低电平时，MCE/PDEN端输出MCE信号。MCE信号是在INTA周期的T_1状态期间有效的信号，可作为主中断控制器8259A的级联地址输出到地址总线时的同步信号使用

3.3 总线仲裁器8289

8289芯片是在多总线（或者与此相同的方式）中裁决总线使用权的总线仲裁器。如果所有的系统都装在一块印制板上，或者系统只有一个处理器，则不需要这种芯片。图3.8、图3.9分别为总线仲裁器8289的引脚连接图和方框图。

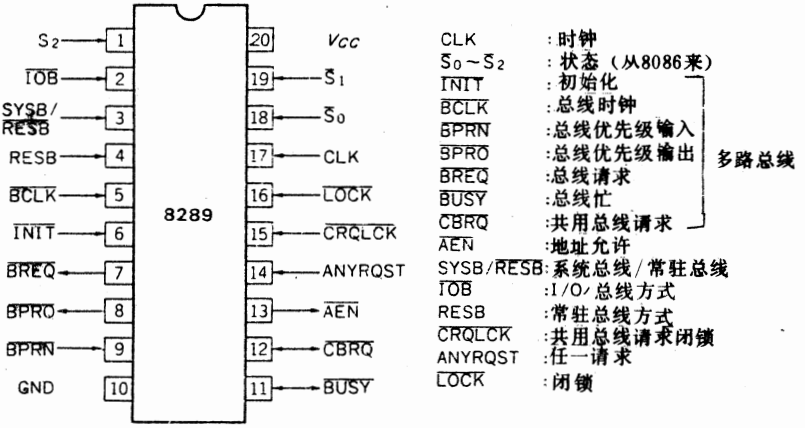


图3.8 8289的引脚连接图

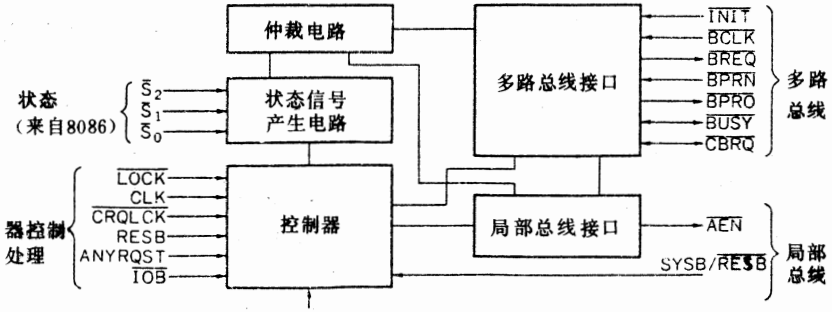


图3.9 8289的内部电路方框图

芯片中的状态信号产生电路，对从8086来的状态进行译码，产

生芯片内容所需要的信号。仲裁电路、多总线接口电路通过多总线与其他模板上的处理器进行裁决。$\overline{\text{INIT}}$、$\overline{\text{BCLK}}$、$\overline{\text{BREQ}}$、$\overline{\text{BPRN}}$、$\overline{\text{BPRQ}}$、$\overline{\text{BUSY}}$和$\overline{\text{CBRQ}}$信号都是向总线输入或者从总线输出的信号。

8289芯片有四种工作方式，由$\overline{\text{IOB}}$、RESB端的信号指定($\overline{\text{IOB}}$端与8288的IOB端可以独立使用）。表3.1列出了四种工作方式中总线的請求、释放条件。8289（如果ANYRQST为“低”电平时）即使对总线的存取已经结束，只要不满足表3.1所列的释放条件也不会释放总线。四种方式具有下述特点，根据模板的不同特性选择不同的方式。

（1）单一总线方式

在模板上只有处理器，而沒有存储器和I/O时采用这种方式。这时，8289在所有的总线存取周期都产生总线請求信号。

（2）I/O总线方式

在模板上只有I/O电路，处理器不存取多总线上的I/O时，采用这种方式。这时，8289只在存储器存取周期产生总线請求信号。

（3）常驻总线方式

这种方式是最普通的一种方式，在模板上有存储器和I/O电路，而且处理器在模板上和在多总线上双方存取存储器和I/O时，使用这种方式。这时，8289在SYSB/$\overline{\text{RESB}}$输入端为“高”时，在总线存取周期請求总线。

（4）I/O总线·常驻总线方式

在模板上有存储器和I/O电路，处理器不存取系统上的I/O时，采用这种方式。这时，8289在SYSB/$\overline{\text{RESB}}$输入端为“高”时，在存储器存取周期請求总线。

在8289为常驻总线方式时，如果采用外部译码器来区别存储器、I/O的存取及内部总线（常驻总线）、多总线（系统总线）的存取，并将信号输入到SYSB/$\overline{\text{RESB}}$端时，则用一种常驻总线方式就可以实现所有的方式。$\overline{\text{IOB}}$、RESB输入信号就是为了减轻这

表3.1 8289的总线请求、释放条件

状态		I/O总线方式	常驻总线方式		I/O总线、常驻总线方式		单一总线方式
命令	$\overline{S_2}\ \overline{S_1}\ \overline{S_0}$	$\overline{IOB}$ = L RESB = L	$\overline{IOB}$ = H RESB = H		$\overline{IOB}$ = L RESB = H		$\overline{IOB}$ = H RESB = L
		—	S/$\overline{R}$ = H	S/$\overline{R}$ = L	S/$\overline{R}$ = H	S/$\overline{R}$ = L	—
中断允许	000		○				○
I/O读	001		○				○
I/O写	010		○				○
暂停	011						
取指令	100	○	○		○		○
存储器读	101	○	○		○		○
存储器写	110	○	○		○		○
无效	111						

表中○：请求总线　S/R：SYSB/$\overline{RESB}$

方　式	控制输入		请求条件	释放条件
	$\overline{IOB}$	RESB		
单一总线方式	H	L	所有的总线周期	TI·($\overline{CBRQ}$ = L)+($\overline{BPRN}$ = L) +HLT
常驻总线方式	H	H	(S/$\overline{R}$ = H)·(总线周期)	(TI+(S/$\overline{R}$ = L))·($\overline{CBRQ}$ = L) +($\overline{BPRN}$ = L) +HLT
I/O总线方式	L	L	所有的存储器周期	((I/O周期) +TI)·($\overline{CBRQ}$ = L) + ($\overline{BPRN}$ = L)+HLT
I/O总线·常驻总线方式	L	H	(S/$\overline{R}$ = H)·(存储器周期)	((I/O周期) +TI+(S/$\overline{R}$ = L)) ·($\overline{CBRQ}$ = L) +($\overline{BPRN}$ = L) +HLT

表中　S/$\overline{R}$：SYSB/$\overline{RESB}$ ；TI：无效状态；HLT：暂停状态

注：· 无效状态、暂停状态没有总线请求。

· ($\overline{BPRN}$ = L) 表示优先权高的主处理器请求总线。

· SYSB/$\overline{RESB}$ 为表示处理器是存取多总线（系统总线）还是存取模板内部的总线（常驻总线）的输入信号。

些译码器而设置的。

8289芯片上，除了上述的引线端之外，还有下面三个引线端，用以控制总线的获得、释放条件。

a) $\overline{\text{LOCK}}$

$\overline{\text{LOCK}}$是锁定已获得的总线的信号，在该端信号为低电平时，8289在所有条件下都不释放总线。通常它与8086的$\overline{\text{LOCK}}$输出相连接（参阅1.5.3）。

b) CRQLCK

CRQLCK是锁定CBRQ信号的判优信号。在CRQLCK为低电平时，就不再考虑表3.1所列的释放条件中的CBRQ项了。CBRQ项是为了使总线使用权优先级低的处理器容易获得总线而设置的。当CRQLCK为低电平时，优先级低的处理器就很难获得总线。

c) ANYRQST

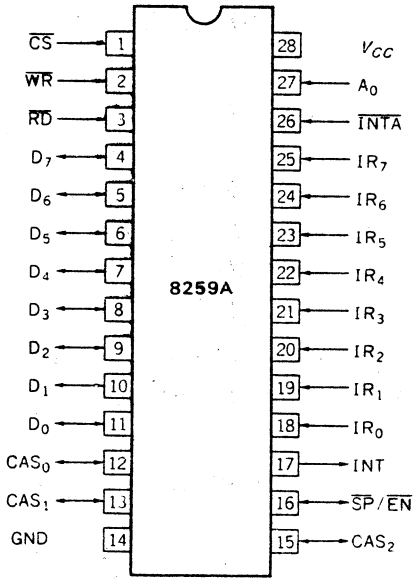
在ANYRQST为低电平时，只要不满足表3.1中的释放条件，即使8289的总线存取已经结束，也不放弃总线。但是，当ANYRQST为高电平时，只要8289的总线存取结束（$\overline{S_0}$～$\overline{S_2}$变成无效时），就释放总线。这样，优先级低的处理器就会容易获得总线，不过反过来却会造成总线的获得、释放频繁交替，使总线的使用效率下降。

3.4 中断控制器8259A

8259A芯片是为了对8085A及8086进行中断控制而设计的芯片，它是可以进行程序控制的中断控制器。8259A不仅可以控制8085A、8086的中断，而且通过级联连接，最多可以控制64级中断请求。各种方式的设定是在初始化时通过软件进行的。

3.4.1 概述

图3.10、图3.11分别是8259A的引脚连接图和方框图。每一次中断处理包括下述过程。



$\overline{CS}$	: 片选
$\overline{RD}$	: 读命令
$\overline{WR}$	: 写命令
A_0	: 地址
$\overline{INTA}$	: 中断允许
INT	: 中断请求
$D_0 \sim D_7$	: 数据总线
$IR_0 \sim IR_7$	: 中断请求输入
$CAS_0 \sim CAS_2$	: 级连地址输出
$\overline{SP}/\overline{EN}$	: 从控制器/程序控制/允许

图3.10 8259A 的引脚连接图

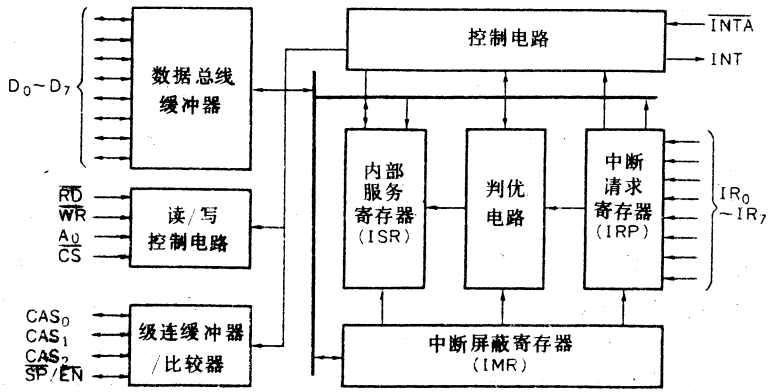


图3.11 8259A 的方框图

（1）在中断請求输入端IR_0～IR_7上产生中断請求。

（2）中断請求锁存在中断請求寄存器IRR中，并与中断主寄存器IMR相“与”，送给优先级判定电路。

（3）优先级判定电路检出优先级最高的中断請求位，并设置该位的内部服务寄存器ISR。

（4）控制电路接受中断請求，并向8086输出INT信号。

（5）8086接受INT信号，进入连续两次的INTA周期。

（6）如果8259A是作为主控的中断控制器设置的，则在第1个INTA周期它就把级联地址从CAS_0～CAS_2送出。

（7）在8259A单独使用时，或是由CAS_0～CAS_2选择的从属中断控制器，就在第2个INTA周期，将类型向量输出给数据总线。

（8）8086读取类型向量，转移到相应的中断处理程序（参阅1.4）。

（9）中断的结束是通过向8259A送一条EOI（中断结束）命令，使ISR复位来实现的。

如果8259A设置为8085A方式，则有3个INTA周期，在这三个INTA周期中输出调用指令。

3.4.2 级联连接法

在单独使用8259A时，如图3.12所示，在IR_0～IR_7上输入中断請求，INT和$\overline{INTA}$与CPU相连接。这时，中断請求输入共计有IR_0～IR_7 8个级别。

8259A可以进行级联连接。图3.13为8259A的级联连接方法。在级联连接中，把1个8259A作为主控制器芯片，在该芯片的IR端连到从属控制器8259A的INT输出端。因此1个主控制器最多可以连接8个从属控制器，中断請求输入为$8\times 8=64$级。

在第1个INTA周期，级联地址CAS_0～CAS_2从主控制器输出，由该级联地址选择的从属控制器在第2个INTA周期输出方式向量。

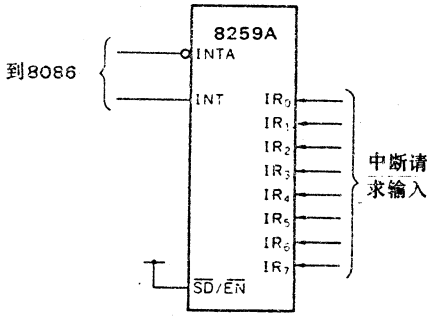


图3.12　8259A单独使用情况

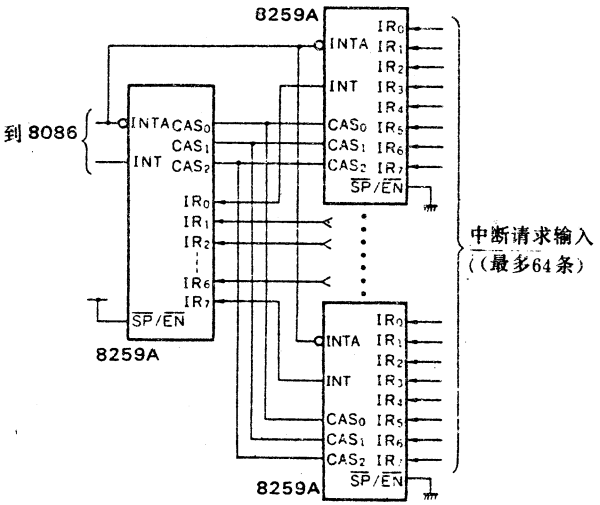


图3.13　8259A的级联连接

主控制器，从属控制器的指定及从属控制器的识别地址，可以通过软件（一部分通过 $\overline{SP}/\overline{EN}$端）进行编程。

没有连接从属控制器的主控制器的IR输入端，可以直接作为中断請求输入端使用。连接或不连接从属控制器也可以通过软件编程来指定。

3.4.3 优先级电路

图3.14为优先级电路的电路结构。

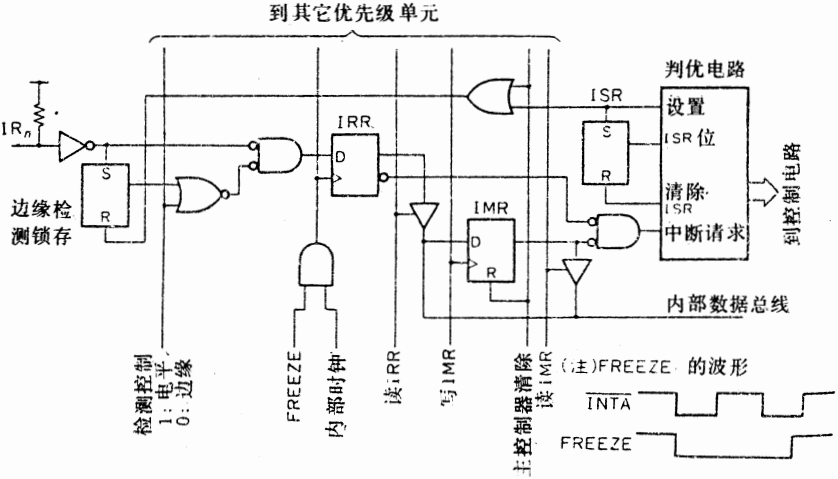


图3.14 8259A的优先级电路

IR端输入的中断請求信号与边缘检测锁存器的内容相“与”，作为IRR的输入信号。边缘检测锁存器用于检出IR输入信号的上升沿。在电平检测时，就不进行上面提到的“与”操作。（边缘检测方式，即需要边缘、也需要电平，这一点与8085A的RST7.5输入的检测方法不同。）

IRR电路对中断請求与内部时钟进行同步取样。IRR电路的输出信号与IMR的内容相“与”后，作为优先级判优电路的输入。通过FREEZE信号使IRR的内容在INTA周期固定不变。或者说，在最初的 $\overline{\text{INTA}}$ 信号输出之前，IR输入信号必须保持高电平。

优先级判优电路检出中断請求中优先级最高的位，设置该位的ISR寄存器，即表示该位处于中断服务中。此后，控制电路将INT

信号输出给8086，并进入INTA周期。

在中断服务过程中，在EOI命令使ISR寄存器复位之前，不再接受优先级低的中断请求。如果有优先级更高的中断请求，将产生多重中断。

IRR、ISR、IMR的内容可以从8086微处理器读出。

IR_0～IR_7的优先级，通常按$IR_0>IR_1>\cdots\cdots>IR_7$的顺序，但是通过程序也可以改为循环方式。

3.4.4 INTA周期

图3.15是INTA周期的波形图。表3.2列出了INTA周期中使用的信号。

（1）使用单个控制器时

这时可以不考虑第1个INTA周期。在第2个INTA周期中，将方式向量输出给数据总线，在缓冲方式时，$\overline{EN}$端为低电平。

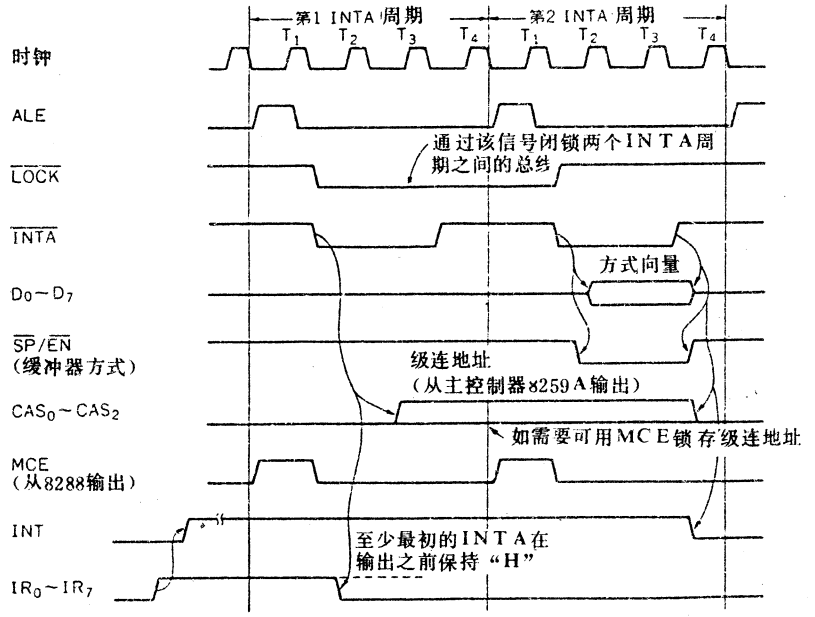


图3.15 INTA周期的波形图

表3.2 在INTA周期中8259A的输入输出

方式	INTA周期	CAS_0～CAS_2	方式向量	$\overline{EN}$（注2）
单个	第1	（注1）		
	第2		输出	输出
级联（主控）	第1	输出		
	第2			
级联（从属）	第1			
	第2	输入	输出	输出

（注1）空白表示无操作。

（注2）EN只在缓冲方式时输出。

（2）级联连接的主控制器

在第1个INTA周期中，将IR_0～IR_7中某一个正在服务的信号，作为级联地址输出给CAS_0～CAS_2。第2个INTA周期不考虑。MCE（主控制器级联允许）信号是从8088输出的信号，必要时，采用该信号锁存级联地址。

（3）级联连接的从属控制器

可不考虑第1个INTA周期。在第2个INTA周期中，读取级连地址，如果该地址与预先编程的内容（地址）一致，则把方式向量输出给数据总线，缓冲方式时，$\overline{EN}$端为低电平。

3.4.5 8259A的程序设计

8259A是可以编程的中断控制器，根据需要可通过软件编程。8259A的命令包括初始设定的ICW（初始化命令字）和操作时的OCW（操作命令字）。这些命令的地址是任意设定的，这一点必须注意。

（1）ICW

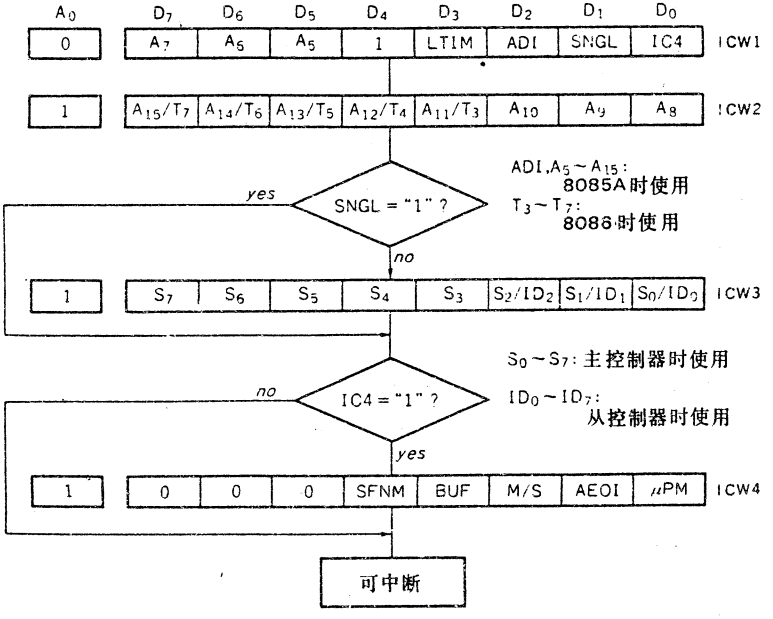


图3.16 8259A的ICW设置过程

图中：

LTIM：中断检测，1＝电平检测，0＝边缘检测。

ADI：CALL地址间隔，1＝4，0＝8。(8086方式时不考虑)。

SNGL：1＝单独使用，0＝级联时使用。

IC4：1＝需要ICW4，0＝不需要ICW4。(8086方式时总是1)。

$A_5 \sim A_{15}$：CALL地址高11位。(在8086方式中只有$T_3 \sim T_7$有效)。

$T_3 \sim T_7$：中断方式向量的高5位。低3位放入$IR_0 \sim IR_7$编码后的值)。

$S_0 \sim S_7$：$IR_0 \sim IR_7$上连接从属控制器8259A时，相对应的位为“1”。

$ID_0 \sim ID_2$：从属控制器的识别地址。(存放$IR_0 \sim IR_7$编码后的值)。

SFNM：1＝特殊完全嵌套方式。

0＝完全嵌套方式。

AEOI：1＝自动EIO方式。

0＝EOI方式。

μPM：1＝8086方式，0＝8085A方式。(无ICW4时，为8085A方式)。

BUF：1＝缓冲器方式，0＝不是缓冲器方式。

M/S：1＝主控制器，0＝从属控制器。

ICW是初始设定时采用的命令。包括ICW1～ICW4四种命令。

图3.16表示命令设置的过程。无论8259A处于什么状态，只要命令的第4位写“1”，第A_0位写“0”，就是ICW1命令。而且将下面的1～3字节的命令作为ICW2～ICW4，进入初始设定操作。

ICW1、ICW4指定8259A的方式。如果在ICW1中，IC4=0，则不再设置ICW4，这与ICW4的各位都置“0”的情况相同。在使用8086方式时，需要把IC4的“0”置为“1”，所以说8086方式、一定要有ICW_4命令。图中写的特殊完全嵌套方式、自动EOI方式、缓冲方式的意义如下：

a）特殊完全嵌套方式

8259A在通常的方式（完全嵌套方式）时，从IR输入端接受一次中断之后，在EOI命令使ISR寄存器复位以前，IR输入端就不再接受后面的中断请求了。但是，当8259A作为主控制器使用时，在对一个从属控制器来的中断进行服务的过程中，还必须能够对同一个从属控制器上另外的IR输入端来的中断请求进行服务。特殊完全嵌套方式就是为了实现多重中断的方式。

b）自动EOI方式

在这种方式中不需要EOI命令，8259A在最后的$\overline{INTA}$信号的上升沿自动执行EOI操作。这种方式尽管不需要EOI命令，但是在中断服务过程中，ISR寄存器已经复位，所以有可能产生优先级更低的中断，这一点务必注意。

c）缓冲方式

这种方式将指定8086的数据总线是否进行缓冲。在指定为缓冲方式时，$\overline{SP}/\overline{EN}$端是输出端，如果输出方式向量时，该端为“低”电平。在指定不是缓冲方式时，$\overline{SP}/\overline{EN}$端是输入端，在该端上进行主控制器、从属控制器的识别（参考表3.3）。

$\overline{EN}$输出信号是表示8259A输出方式向量的信号，用于8086的等待信号产生及数据总线缓冲器的控制中。

（2）OCW

OCW是操作过程中给出的命令。初始设定结束后的命令都可

表3.3　缓冲方式的指定

BUF	M/S	$\overline{SP}/\overline{EN}$ 端		主/从属
0	不管	输入	L	从属
			H	主
1	0	输出 $\overline{EN}$		从属
	1			主

ICW4的BUF位为“0”时，主从控制器的识别在 $\overline{SP}/\overline{EN}$ 端进行。BUF 位为“1”时，主/从控制器的识别在 M/S 位上进行，这时 $\overline{SP}/\overline{EN}$ 端为输出端。

看成是OCW命令。在OCW命令中包括OCW1～OCW3三种命令。图3.17表示OCW的各种命令的格式。

OCW1是对IMR置位、复位的命令，在$A_0=1$时就是OCW1命令。

OCW1

A_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
1	M_7	M_6	M_5	M_4	M_3	M_2	M_1	M_0

M_0～M_7：中断屏蔽（“1”=置位，“0”=复位）

OCW2

A_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	R	SL	EOI	0	0	L_2	L_1	L_0

L_1～L_2：ISR位的指定

R SL EOI

0 0 1 ：一般EOI（对正在服务的ISR复位）

0 1 1 ：指定EOI（对L_0～L_2指定的ISR复位）

1 0 1 ：执行一般EOI，该位的优先级为最低

1 0 0 ：设置为循环方式（此后每遇EOI，优先级就循环）

0 0 0 ：循环方式复位（此后优先级不循环）

1 1 1 ：执行指定EOI，该位的优先级最低

1 1 0 ：不执行EOI，L_0～L_2指定的位优先级最低

0 1 0 ：无操作

OCW3

A_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	X	ESMM	SMM	0	1	P	RR	RIS

ESMM SMM

0 × ：无操作

1 0 ：特殊屏蔽方式复位

0 1 ：特殊屏蔽方式置位

P RR RIS

0 0 × ：无操作

0 1 0 ：在下一读指令时读IRR

0 1 1 ：在下一读指令时读ISR

1 × × ：查询命令（在下一读指令读中断状态）

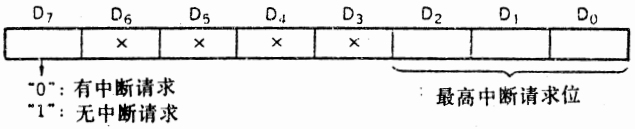


图3.17 OCW命令的格式

OCW2是EOI命令，用于指定复位的ISR寄存器及改变优先级。在采用单个8259A时，通常通过EOI命令，使正在服务的位所指示的ISR寄存器复位。

OCW3是指定设置特殊屏蔽方式和读内部寄存器的命令。也可以在写入OCW3后，接着再以$A_0=0$读出，这时再读内部寄存器。（在以$A_0=1$读时，任何时候都可以读IMR的内容。）

在通常的方式时，中断服务过程中，ISR设置期间，对优先级更低的中断請求并不响应。特殊屏蔽方式就是可以解除这种禁止中断状态的方式。在这种方式时，除了由ISR设置的位和由IMR屏蔽的位表示的中断外，其他所有级别的中断都可响应。

操作过程中，以$A_0=1$写命令时，8259A根据第3位、第4位内容的不同，按下述情况对命令进行区分。

第4位	第3位	命令
1	×	OCW_1
0	0	OCW_2
0	1	OCW_3

4 硬件设计

8086微处理器的硬件结构有下述特点，在进行系统的硬件设计时，必须充分了解这些特点，并依据这些特点进行硬件设计。

- 20位地址总线
- 16位数据总线
- 按字节（8位）进行地址分配
- 可以8位为单位进行读、写
- 多路地址总线和数据总线
- 系统有最小和最大两种组成方式

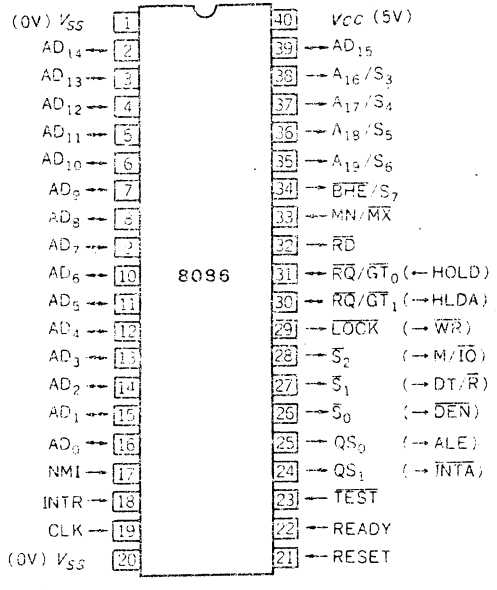
4.1 引脚连接图和各信号的功能

图4.1、图4.2分别为8086和8288芯片的引脚连接图。系统的组成是最小方式时，所有的控制信号都从8086芯片输出，而最大方式时，几乎所有的总线控制信号都从8288芯片输出。表4.1列出了最小方式和最大方式时，输出信号的差别。必须注意，即使名称相同的信号，在方式不同时，输出的波形也可能不同。

4.1.1 存储器周期

8086微处理器在进行存储器、I/O访问时，一个周期由T_1～T_4 4个状态组成，称为总线周期。因为8086微处理器内部分成执行部分（EU）和总线接口部分（BIU），它对存储器，I/O的访问和内部的执行是各自独立的，所以不再称机器周期。

图4.3、图4.4表示了8086的最小方式、最大方式的存储器周期



（ ）内为最小方式时的引脚名

图4.1 8086的引脚连接图

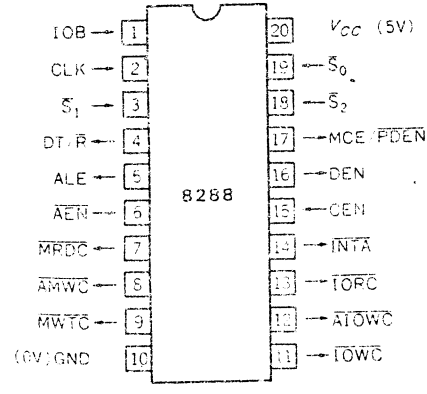


图4.2 8288的引脚连接图

表4.1 最小方式和最大方式输出信号的差别

信　号	最 小 方 式	最 大 方 式
DT/$\overline{R}$	8086脚27	8288脚 4
$\overline{DEN}$	8086脚26	8288脚16 以正逻辑DEN输出
ALE	8086脚25	8288脚 5
$\overline{INTA}$	8086脚24	8288脚14
M/$\overline{IO}$	8086脚28	如锁存$\overline{S}_2$，可使用M/$\overline{IO}$
命　令	以$\overline{RD}$、$\overline{WR}$、M/$\overline{IO}$形式从8086输出 采用LS257可改变为（$\overline{MRDC}$、$\overline{MWTC}$、$\overline{IORC}$、$\overline{IOWC}$）	以$\overline{MRDC}$、$\overline{MWTC}$、$\overline{IORC}$、$\overline{IOWC}$、$\overline{AMWC}$、$\overline{AIOWC}$的形式从8288输出
HLDA、HOLD $\overline{RQ}$/$\overline{GT}_0$、 $\overline{RQ}$/$\overline{GT}_1$	8086的脚30、31分别作为HLDA、HOLD使用	8086的脚30、31分别作为$\overline{RQ}$/$\overline{GT}_1$、$\overline{RQ}$/$\overline{GT}_0$使用
只在最大方式输出		$\overline{LOCK}$、$\overline{S}_0$、$\overline{S}_1$、$\overline{S}_2$ QS_0、QS_1

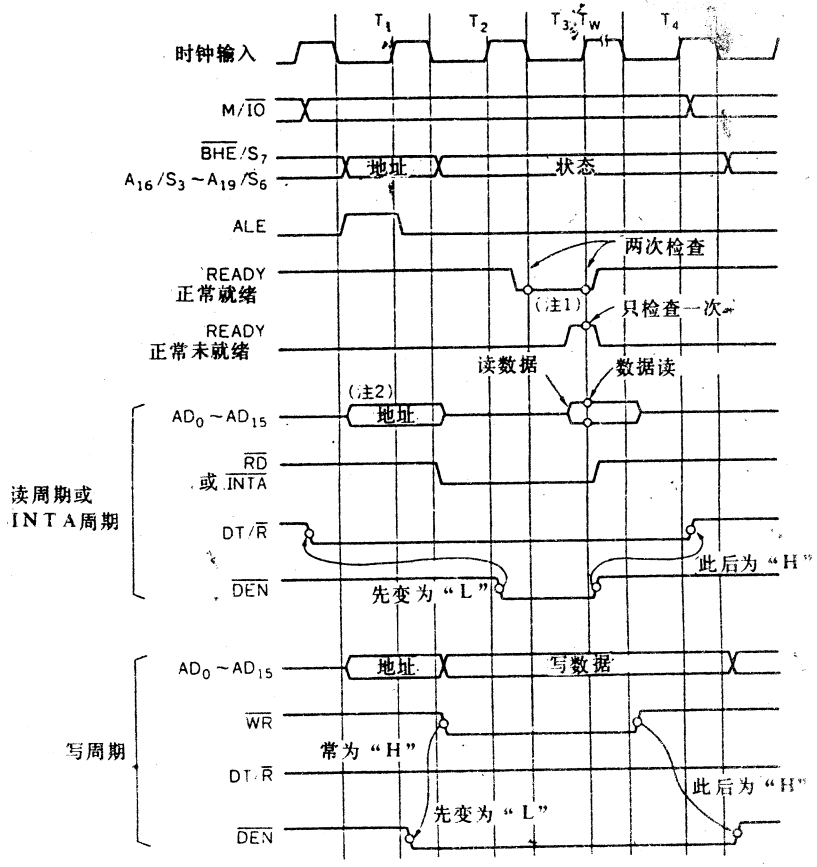
和各信号的波形图。T_1～T_4的各状态从时钟的下降沿开始，到下一个时钟的下降沿结束。等待状态T_W夹在T_3和T_4之间。

8086微处理器只在需要读/写数据时，才对总线进行存取，所以，总线周期并不一定隔4个状态才开始，而是在一个总线周期结束以后n（n＝2～∞）个状态的无效周期后开始下一个总线周期。

4.1.2 最小方式和最大方式中的共同信号

（1）AD_0～AD_{15}（输入/输出、三态）

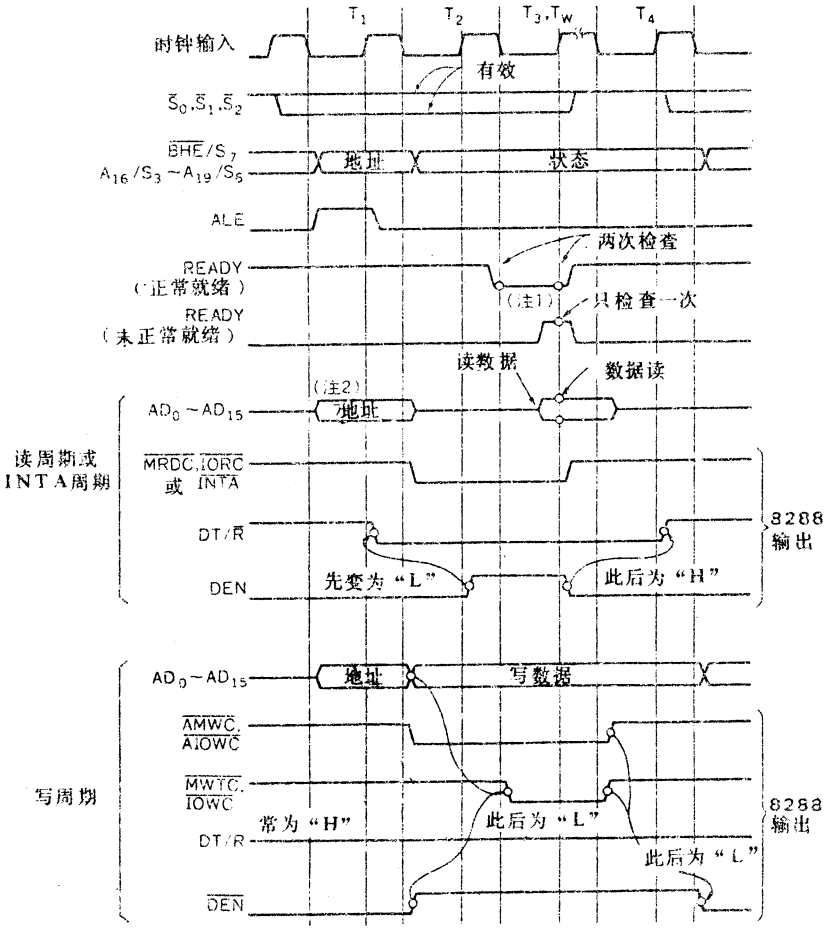
AD_0～AD_{15}是16位数据总线和20位地址总线中的低端16位信



（注1）READY在T_2的结尾及T_3、T_W的时钟上升沿采样。

（注2）在INTA周期不输出地址，在第2次INTA周期之前，AD总线为浮动状态。

图4.3 最小方式的波形



（注1）READY在T_2的结尾及T_3、T_W的时钟上升沿采样。

（注2）在INTA周期不输出地址，在第2次INTA周期之前AD总线为浮动状态。

图4.4 最大方式的波形图

号。由于二者是分时输出的，所以地址总线需要通过LS373、8283等芯片进行锁存。地址总线信号在T_1状态输出，T_2～T_3状态，对读周期来说是浮动状态，而对写周期则输出写数据。在INTA周期中，地址总线是浮动状态。（在AD_8～AD_{10}上可以输出8259A的级联地址。）

（2）A_{16}/S_3～A_{19}/S_6（输出、三态）

A_{16}～A_{19}是地址总线的高端4位信号，S_3～S_6是状态信号。二者以分时输出。即在T_1状态输出地址，在T_3～T_4状态输出状态信号。状态信号中的S_3、S_4表示在总线周期中访问段寄存器的情况（参考表4.2）。S_5表示中断允许标志的内容，S_6总是处于低电平。当访问I/O时，A_{16}～A_{17}均输出低电平。

（3）$\overline{BHE}/S_7$（输出、三态）

$\overline{BHE}$信号在T_1状态时输出，S_7信号在T_2～T_4状态时输出。关于$\overline{BHE}$信号請参考1.1.3。S_7是备用的状态信号，其内容不固定。

表4.2 S_3、S_4和使用的段寄存器

S_4	S_3	使用的段寄存器
0	0	数据段寄存器
0	1	堆栈段寄存器
1	0	代码段寄存器或不使用（注）
1	1	附加段寄存器

0为L；1为H

（注）I/O存取、INTA周期等

（4）READY（输入）

READY信号是表示数据传送已结束的信号。8086在T_2的结束和T_3或T_W的时钟上升沿两个地方对READY信号取样。这样，就产生图4.5所示的两种等待的情况。

8086的READY信号，有正常就绪和非正常就绪两种状态，分

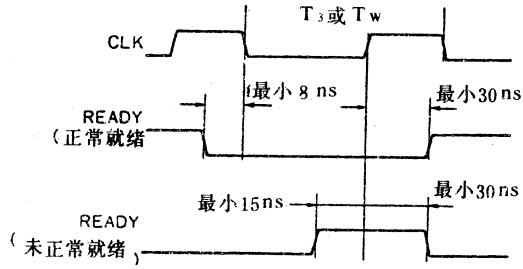


图4.5 READY信号的波形图

别满足图中的时间关系（时钟频率为5MHz）。正常就绪型，通常是把READY输入置于高电平（就绪），只在存取需要等待的设备时，才变为低电平。正如图4.5所示，这种方式从T_2状态的结束到Tw状态的时钟上升沿必须保持低电平。

非正常就绪型，通常是把READY输入置于低电平(未就绪)，只在存取设备上需要定时时才变为高电平。由于这种方式将READY输入的取样延长到T_3、Tw状态的时钟上升沿，因此它对动态RAM（通过刷新等待时）那种与存取时间不固定的器件进行接口是很有利的。

无论哪一种就绪的方式，READY信号都必须满足图4.5所示的建立时间和保持时间。如果不满足这个定时关系，比如如图4.6所示的错误定时关系时，由于READY信号在T_2结束时为高电平(就绪)，T_3的上升沿为低电平（未就绪），所以即使输入READY信号，8086也不能正常工作。

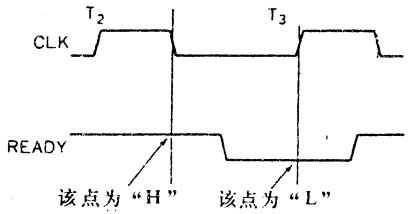


图4.6 错误的READY信号波形

（5）CLK（输入）

CLK是时钟信号的输入端。通常与8284A的时钟输出相连接。时钟信号的脉冲宽度低和高的比通常采用2:1（占空度为1/3）。

（6）RESET（输入）

RESET为复位输入信号。通常与8284A的复位输出端相连接。复位脉冲宽度最小需要4个时钟。在接通电源时不能小于50μS。

表4.3列出了复位后内部寄存器的状态，表4.4是复位时各端的状态。表中从$\overline{S_0}$/（$\overline{DEN}$）到$\overline{INTA}$各端均处于浮动状态，因此在这些端上需要连接上拉电阻。（8288的$\overline{S_0}$～$\overline{S_1}$输入端上连接有上拉电阻。）

表4.3 复位后内部寄存器的状态

内部寄存器	内容
标志寄存器	清除
IP	0000_{16}
CS寄存器	$FFFF_{16}$
DS寄存器	0000_{16}
SS寄存器	0000_{16}
ES寄存器	0000_{16}
指令队列寄存器	清除

（7）NMI（输入）

NMI信号是非屏蔽的中断输入信号。当NMI信号输入为高电平时，就产生类型2的中断（参见1.4）。虽然输入是边沿检测的，但为了使操作可靠，在两个时钟期间保持低电平之后，再保持两个时钟的高电平。

（8）INTR（输入）

INTR是中断请求输入端。8086微处理器在各指令的最后时钟周期，对INTR的输入端进行取样，如果是高电平，就进入INTA周期（参阅1.4）。在第1个$\overline{INTA}$输出信号输出之前，INTR输入

表4.4 复位时输出端的状态

引 脚 名	状 态
$AD_0 \sim AD_{15}$	浮 动
$A_{16}/S_{16} \sim A_{19}/S_6$	浮 动
BHE/S_7	浮 动
$\overline{S_0}$/ ($\overline{DEN}$)	输出高电平后浮动
$\overline{S_1}$/ (DT/$\overline{R}$)	输出高电平后浮动
$\overline{S_2}$(M/$\overline{IO}$)	输出高电平后浮动
$\overline{LOCK}$/($\overline{WR}$)	输出高电平后浮动
$\overline{RD}$	输出高电平后浮动
$\overline{INTA}$	输出高电平后浮动
ALE	低电平
HLDA	低电平
$\overline{RQ}$ /$\overline{GT_0}$	高电平
$\overline{RQ}$ /$\overline{GT_1}$	高电平
QS_0	低电平
QS_1	低电平

信号必须保持高电平。

（9）$\overline{INTA}$（输出、三态）

$\overline{INTA}$是中断允许信号。在INTA周期中，没有读命令，这时$\overline{INTA}$信号变为有效信号（低电平）（参见图4.3、图4.4）。在第2个INTA周期，8086从$AD_0 \sim AD_7$读取8位的中断方式向量（参阅1.4）。在最大方式时，该信号从8288输出。

（10）ALE（输出）

ALE为地址锁存信号。在最大方式时，从8288芯片输出。

（11）$\overline{DEN}$（输出、三态）

$\overline{DEN}$为数据允许输出信号。在读周期和INTA周期中，读数据的时候，该信号为低有效。而在写周期时，只在数据有效时，该信号才低有效。在8085A微处理器中，这类信号是采用$\overline{RD}$输出信号

和$\overline{WR}$输出信号的“或”信号，而在8086微处理器中采用的是$\overline{DEN}$输出信号。由于它连接在数据总线缓冲器的$\overline{OE}$（输出允许）端，所以可以避免总线的冲突。在最大方式时，该信号以正逻辑DEN从8288输出（参考图4.3、图4.4）。

（12）DT/$\overline{R}$（输出、三态）

DT/$\overline{R}$为数据传送/接收的输出信号。在读周期和INTA周期时，在$\overline{DEN}$变成有效以前，DT/$\overline{R}$变为低电平，而$\overline{DEN}$变成无效以后，DT/$\overline{R}$又返回高电平。由于它与数据总线缓冲器的DIR端相连接，因此可以避免总线的冲突。在最大方式时，该信号从8288输出。

（13）$\overline{TEST}$（输入）

$\overline{TEST}$信号是为了与其他处理器同步执行指令而准备的信号。当执行WAIT指令时，在$\overline{TEST}$信号变成低电平之前，8086要进入等待状态。通常，该信号与8087处理器的BUSY输出端相连接。

（14）MN/$\overline{MX}$（输入）

MN/$\overline{MX}$是指定最小方式或最大方式的信号。在最小方式时该信号与Vcc端相连接，最大方式与Vss端相连接。

（15）Vcc、Vss

这二者均为电源引线端。有两个Vss端，均必须接地。

4.1.3 只在最小方式时使用的信号

（1）M/$\overline{IO}$（输出、三态）

M/$\overline{IO}$信号用于指定存取的设备是存储器还是I/O。该信号从前一个总线周期的T_4状态的时钟上升沿开始有效。

（2）$\overline{RD}$（输出、三态）

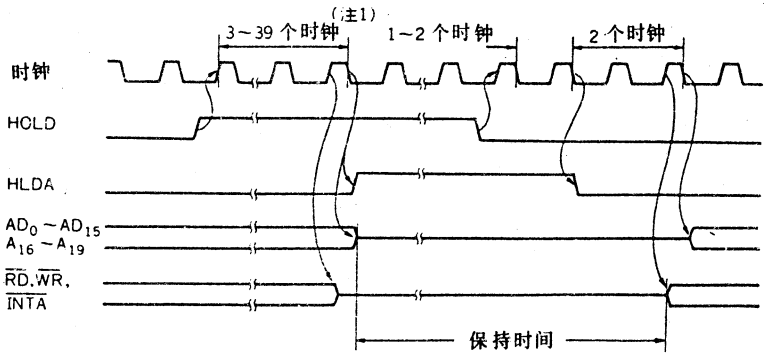
$\overline{RD}$是读数据信号。

（3）$\overline{WR}$（输出、三态）

$\overline{WR}$是写数据信号。

（4）HOLD（输入）

HOLD是保持請求信号。在最小方式时，通过HOLD、HLDA信号裁决与其他处理器之间的总线使用权。图4.7表示HOLD、HLDA信号的波形。



图中：

（注1）从保持信号输入到允许信号输出的潜伏时间如下：

执行中的指令	潜伏时间
无LOCK的指令执行中	3～6(10) 时钟
INTA周期内	15 时钟
LOCK XCHG 指令执行中	24～29(39) 时钟

（ ）内的值为存取从奇数地址开始的字时

（注2）最小方式中无 $\overline{\text{LOCK}}$ 端，但如在指令上加LOCK前缀时，则该指令执行结束之前不输出HLDA信号。

图4.7 HOLD、HLDA信号的波形图

（5）HLDA（输出、三态）

HLDA是保持允许信号。

在最小方式时没有$\overline{\text{LOCK}}$端。如果在指令中加LOCK前缀，则在该指令执行结束之前，不输出HLDA信号。

4.1.4 只在最大方式时使用的信号

（1）$\overline{S_0}$～$\overline{S_2}$（输出、三态）

$\overline{S_0}$～$\overline{S_2}$信号是表示该总线周期存取哪种设备的状态信号。在前一总线周期的T_4状态的时钟上升沿开始有效，在本总线周期的T_4状态之前的状态（如果没有Tw状态，就是T_3状态）时变为无效。

在最大方式时，全部总线控制信号均从8288芯片输出。这时，$\overline{S_0}$～$\overline{S_2}$信号就是为了向8288传送总线时钟种类和定时波形的状态信号。

表4.5列出了$\overline{S_0}$～$\overline{S_2}$信号所表示的总线周期的种类。

表4.5 $\overline{S_0}$～$\overline{S_2}$信号与总线周期

$\overline{S_2}$ $\overline{S_1}$ $\overline{S_0}$	总线周期
000	INTA周期
001	I/O读周期
010	I/O写周期
011	暂停
100	取指令周期
101	读存储器周期
110	写存储器周期
111	无效

S_2可以用于判断存储器、I/O的存取。

（2）QS_0、QS_1（输出）

QS_0、QS_1信号在每个时钟信号的上升沿都输出，它表示在该时钟周期中队列缓冲器存取的状态，如表4.6所示。

（3）$\overline{RQ}/\overline{GT_0}$、$\overline{RQ}/\overline{GT_1}$（输入/输出、漏极开路）

表4.6 QS_0，QS_1信号与队列状态

QS_0 QS_1	队列状态
00	无操作
01	从队列缓冲器中取出指令的第一个字节
10	清除队列缓冲器
11	从队列缓冲器中取出第二个字节以后部分

QS_0、QS_1的内容在每个时钟周期都进行更新

$\overline{RQ}/\overline{GT_0}$、$\overline{RQ}/\overline{GT_1}$信号是在最大方式时裁决总线使用权的信号。最大方式时，在$\overline{RQ}/\overline{GT}$（請求/允许）线上双向进行应答。$\overline{RQ}/\overline{GT}$有两个引脚，即$\overline{RQ}/\overline{GT_0}$和$\overline{RQ}/\overline{GT_1}$，在二者同时有請求时，$\overline{RQ}/\overline{GT_0}$端优先。详细内容請参考1.5.1。

（4）$\overline{LOCK}$（输出、三态）

$\overline{LOCK}$信号是为了在执行某一条指令时，一直独占模板间的总线（系统总线）而准备的信号。在执行带有LOCK前缀的指令过程中，$\overline{LOCK}$信号一直保持低有效。详细内容請参考1.5.2。通常$\overline{LOCK}$信号与8289的$\overline{LOCK}$输入端相连接。

4.1.5 最大方式时从8288输出的信号

在最大方式时，总线控制信号从8288总线控制器输出。总线控制器8288芯片是以多总线为前提而设计的芯片。芯片上的$\overline{MRDC}$、$\overline{IORC}$、$\overline{MWTC}$、$\overline{IOWC}$、$\overline{INTA}$信号的定时波形和直流特性均应满足多总线的规格。

（1）$\overline{MRDC}$、$\overline{IORC}$（输出、三态）

$\overline{MRDC}$、$\overline{IORC}$信号分别为存储器读命令和I/O读命令。在最大方式时，存储器和I/O的有关信号分别以独立的命令信号输出。

（2）$\overline{MWTC}$、$\overline{IOWC}$（输出、三态）

$\overline{MWTC}$、$\overline{IOWC}$分别为存储器写命令和I/O写命令。该信号的

输出信号在T_3状态出现。这一信号波形出现较晚的理由是：它必须等到8086输出的写数据稳定之后才能变为有效。这一点是多总线系统中的必要条件。

（3）$\overline{AMWC}$、$\overline{AIORC}$（输出、三态）

$\overline{AMWC}$、$\overline{AIORC}$分别为提前存储器写命令和提前I/O写命令信号。二者在T_2状态输出。如果用$\overline{MWTC}$、$\overline{IOWC}$命令，写脉冲宽度的最小值不能满足时，就采用$\overline{AMWC}$、$\overline{AIORC}$命令信号。必须注意，在该信号有效之前写数据并未确定。（该信号不能向多总线输出。）

（4）其他信号

其他还有ALE、DEN、DT/$\overline{R}$、$\overline{INTA}$等信号，这些信号請参阅4.1.2的叙述。

4.2 多总线模板的设计举例

本节以多总线兼容模板为例说明8086的硬件设计。图4.8～图4.14是这些模板的方框图和逻辑电路图。为了说明简单起见，例中从多总线发出的中断是非总线向量中断（参阅6.2.4）。

由于8086的硬件结构是以多总线为前提考虑的，所以在这样简单的例子（不采用向量中断和2块RAM板）中，外加的芯片可以很少。

这些电路图中加有斜线（/）的信号表示负逻辑信号。在多总线信号上加有星号*的也是负逻辑信号。

4.2.1 规格与方框图

图4.8为模板的方框图。其规格如下：

（1）CPU　8086　5MHz（最大方式）

（2）ROM　16Kb　2732或2732A×4

（3）RAM　16Kb　M58725×8

（4）串行I/O　2通道　8251A×2

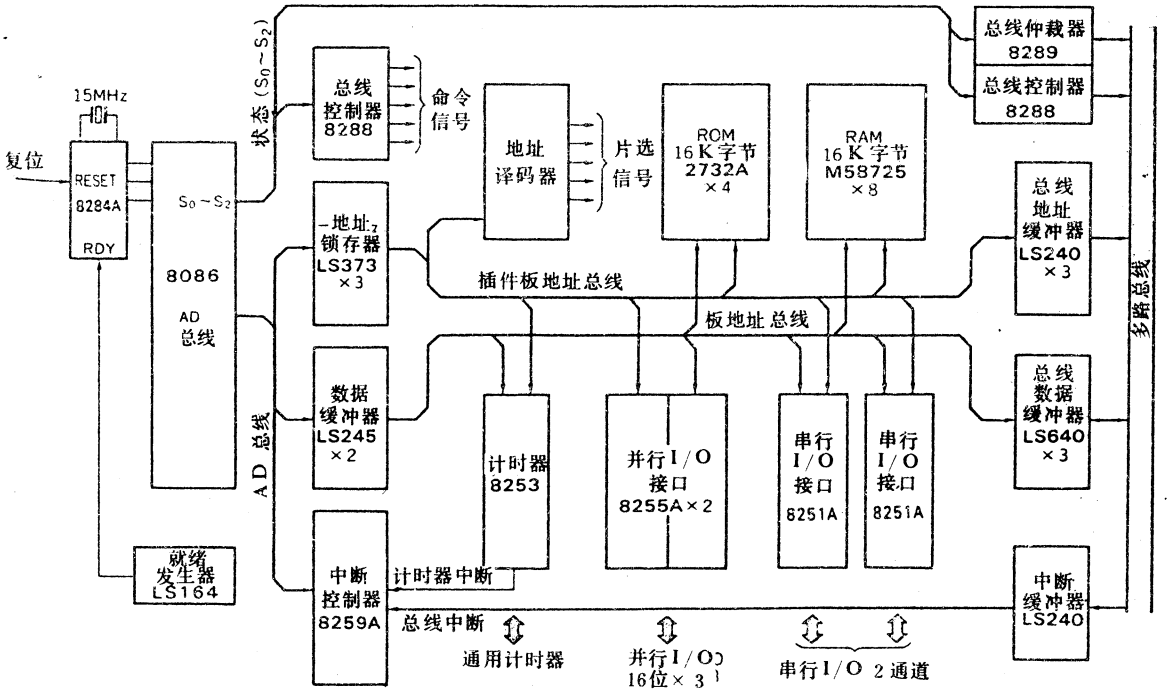


图4.8 多总线板的方框图

（5）并行I/O　16位×3　8255A×2

（6）计时器　16位×3　8253

（7）中断　9级　8259A　（采用非总线向量中断）

4.2.2 CPU

图4.9表示的是CPU的逻辑电路图。8086的系统组成采用最大方式，时钟频率为5MHz。在时钟发生器 8284A 中产生15MHz的时钟信号，经过 3 分频后变成5MHz输入到 8086。复位信号从多总线P_2的AUX RESET端输入。就绪信号由LS164 电路产生，通过无等待的芯片选择信号和 1 等待的芯片选择信号来选择需要的定时信号（参阅图4.10）。8284A的另一系统的就绪电路在存取多总线时使用（直接输入XACK信号）。

地址锁存器采用LS373电路。BHE、$\overline{S_2}$ 也同时锁存，被锁存的$\overline{S_2}$作为 M/$\overline{IO}$信号输出。数据缓冲器采用LS245电路，当直接连接到AD总线上的8259A输出数据时，数据缓冲器将不起作用。因此，在这种情况时，8259A必须用程序设定缓冲器方式（参阅 3.4.5）。

有关总线控制的信号全部从总线控制器8288输出。状态信号中的$\overline{S_0}$～$\overline{S_2}$信号输出给（另外的作为多总线接口用的）8288 和 8289 芯片。写命令采用提前的方式。

4.2.3　地址译码器、中断控制器、计时器

图4.11为地址译码器、中断控制器、计时器部分的逻辑电路图。地址译码器是一个很普通的电路，通过门电路和译码电路产生所需要的片选信号。表4.7列出了译码地址。从表中可以看出，两个8255A分别译码为高位字节和低位字节，而两个 8251A 都译码为低位字节。图4.11中左侧中间的LS74A电路是为了在INTA 周期中，禁止选择I/O、禁止存取多总线而加的电路。

为了使产生等待的电路简单，例中的I/O控制器一律选为“1”等待。（8259A在INTA周期中也可以无等待，例中采用了无等待。）M58725电路不需要等待。2732电路需要“1”等待，而2732A则不

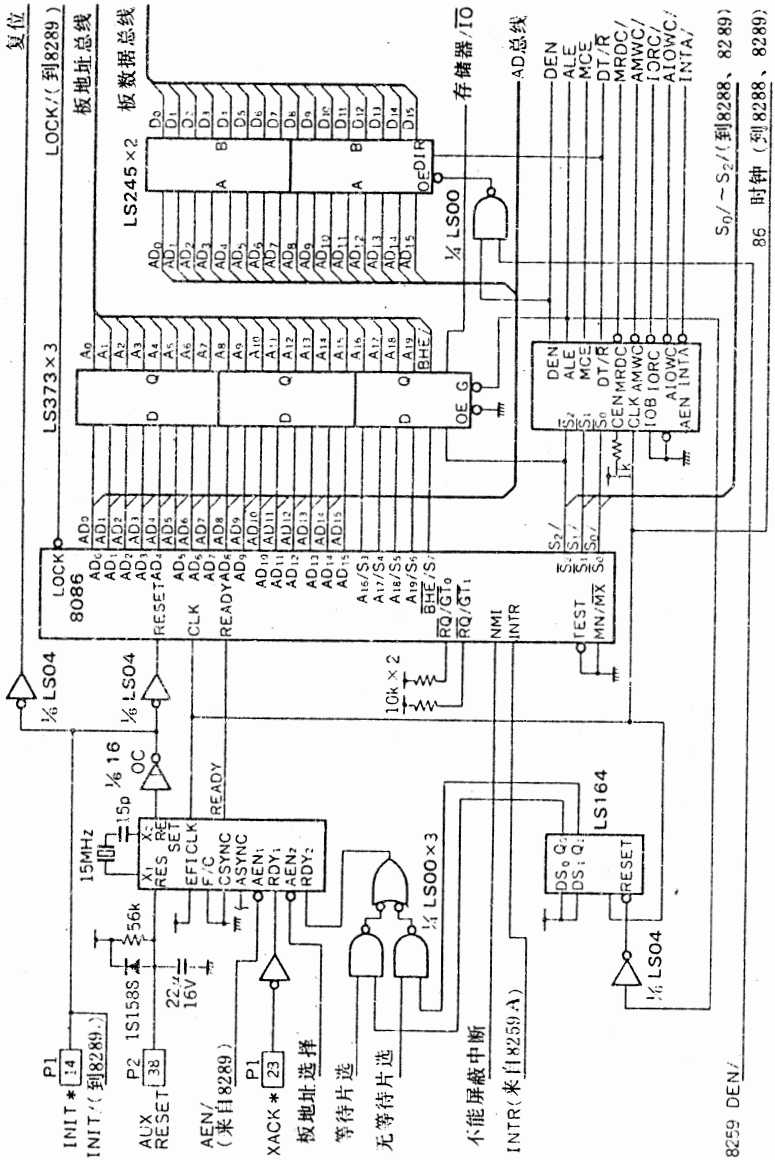


图4.9 多总线板的CPU电路

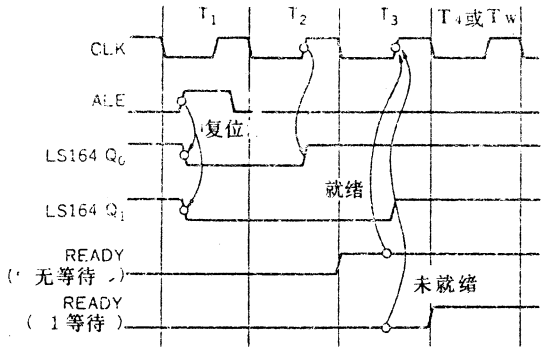


图4.10 LS164产生就绪信号的波形图

需要等待（用跳线进行选择）。

中断控制器采用8259A芯片。中断請求来自多总线，还是来自计时器由跳线进行选择。当8259A是通过 $\overline{RD}$ 、$\overline{INTA}$ 命令将数据输出给数据总线时，SP/$\overline{EN}$ 端为低电平，使连接在AD总线上的数据缓冲器无效。D_0～D_7信号直接连接在AD总线上。

计时器采用8253芯片。在8253芯片内部有3个16位计数器，其中1个计数器的输出作为计时器中断信号输出到8259A。其他2个计数器是通用计数器。计时器时钟来自8251A的晶体振荡器所产生的振荡输出信号。

4.2.4 ROM、RAM

图4.12为ROM、RAM的逻辑电路图。ROM采用4块2732电路，RAM采用8块M58725电路（三菱公司生产的16K静态RAM），二者各为16Kb。

采用LS139电路对RAM进行译码，在低位字节、高位字节分别控制 A_0、$\overline{BHE}$。ROM电路是将A_{13}翻转后译码，由于ROM不需要写入，所以不需要RAM电路中的LS139控制电路。

为了减轻数据总线的负荷，在ROM、RAM的电路中增加了一级数据缓冲器。

4.2.5 串行I/O，并行I/O控制器

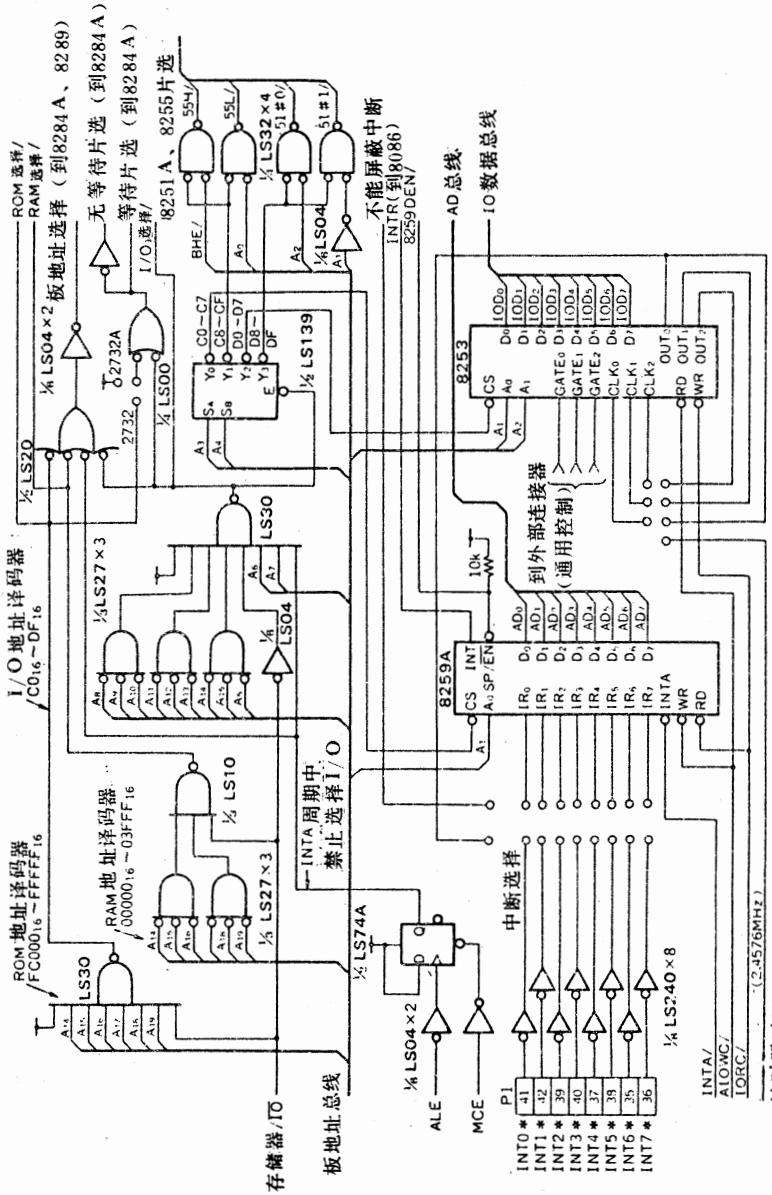


图4.11　多总线板的地址译码器、中断控制和计时器电路

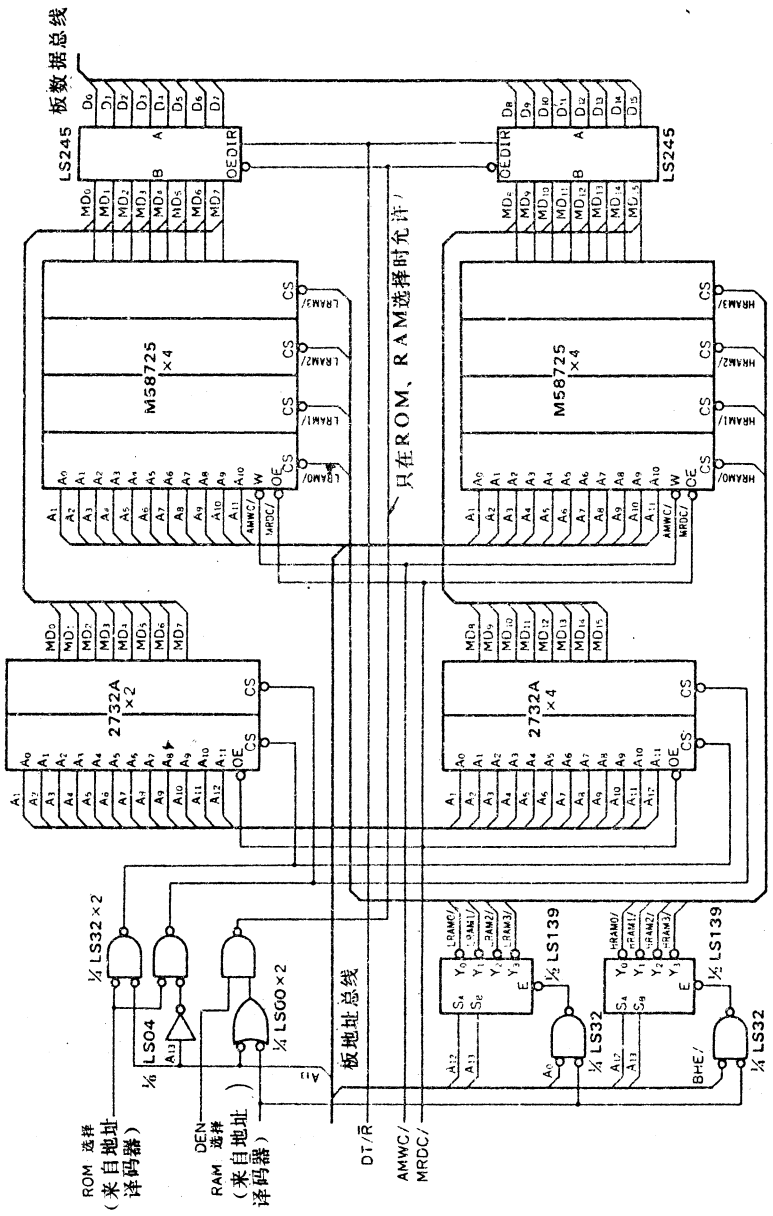


图4.12　多总线板的ROM、RAM部分的逻辑电路图

表4.7　译码地址

器件	译码地址
ROM	00000_{16}～$03FFF_{16}$
RAM	$FC000_{16}$～$FFFFF_{16}$
8259A	$00C0_{16}$～$00C2_{16}$
8255A低位	$00C8_{16}$～$00CE_{16}$
8255A高位	$00C9_{16}$～$00CF_{16}$
8253	$00D0_{16}$～$00D6_{16}$
8251A#0	$00D8_{16}$～$00DA_{16}$
8251A#1	$00DC_{16}$～$00DE_{16}$

（注）I/O控制器的地址相隔1字节

串行I/O控制器采用两个8251A芯片，并行I/O控制器采用两个8255A芯片。为了实现16位的输入输出，并行I/O控制器的两个8255A分别分配在低位字节和高位字节（方式必须同样设定）。两个8251A都分配在低位字节。如图4.13所示。

为了减轻数据总线的负荷，I/O控制器上也加有数据缓冲器。

时钟发生器产生的19.6608MHz的时钟，通过LS393进行分频。2分频后的9.8304MHz时钟作为多总线的BCLK*和CCLK*信号。晶体振荡器，在15MHz以上时使用图中所示的电路。LS393是一个芯片装有2个16进制计数器的电路，这种芯片单独作分频器使用是很方便的。

4.2.6　多总线接口

图4.14是多总线的仲裁器、控制器逻辑电路图。采用8288、8289芯片，可以使与多总线的接口变得很简单。8289芯片在常驻总线方式中使用。

对多总线的存取请求，是通过8289的SYSB端输入模板地址选择信号进行的，即在模板上的器件没被选择时，就可访问多总线。在此之后，8289、8288对状态信号$\overline{S_0}$～$\overline{S_2}$译码，生成所需要的信号。总线$\overline{DEN}$信号、总线DT/$\overline{R}$信号是多总线的数据总线缓冲

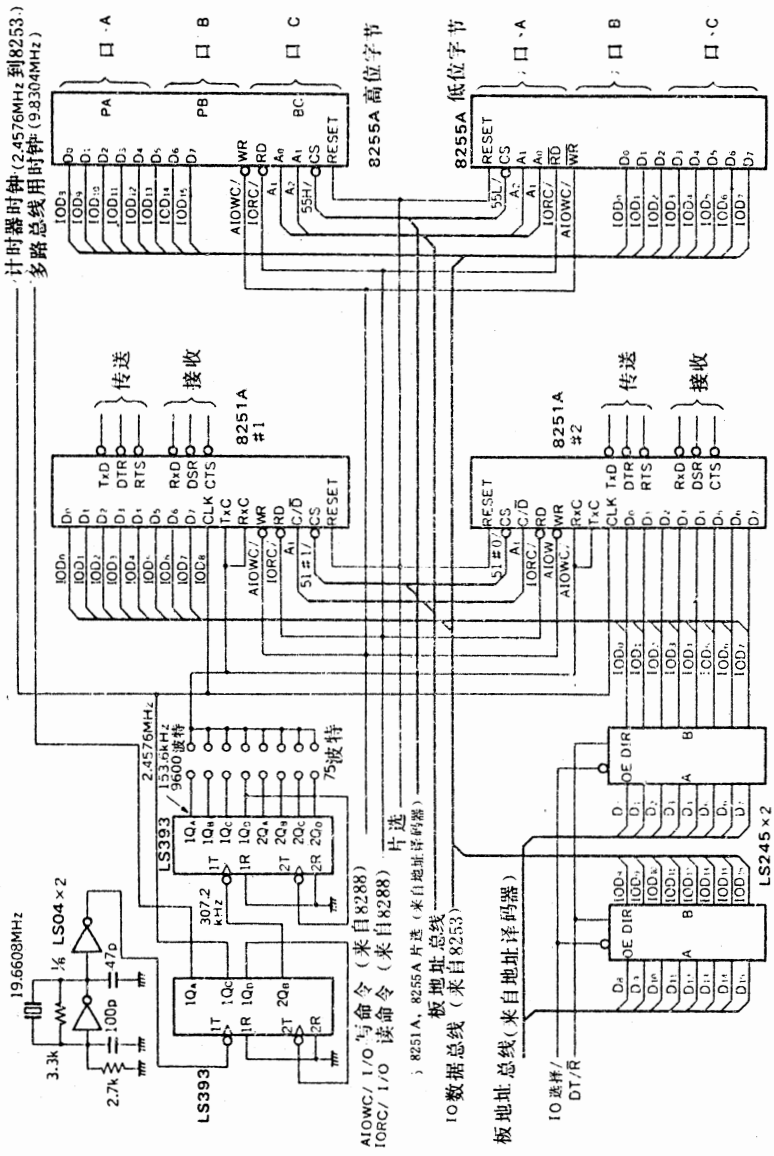


图4.13 多总线板的串行I/O、并行I/O控制器的逻辑电路图

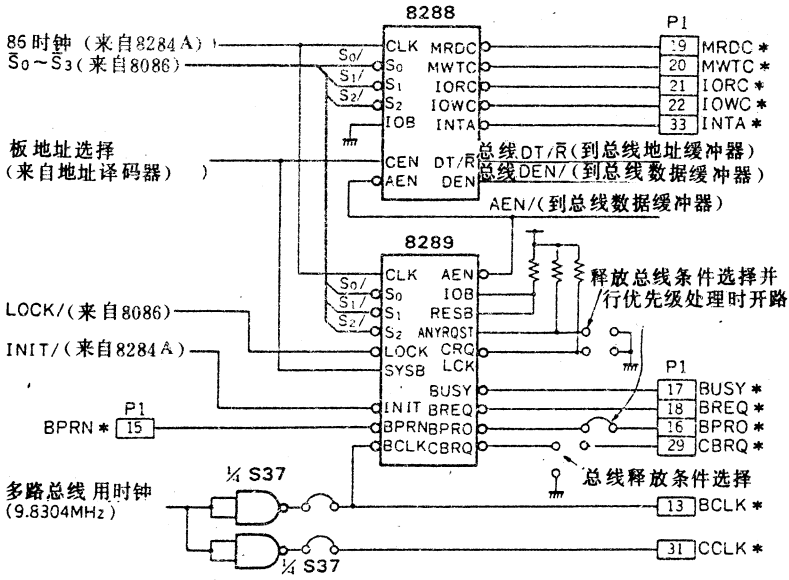


图4.14　多总线板的总线仲裁器、总线控制器的电路图

器中使用的信号。8289右边的跳线，作为总线释放条件的选择使用（参阅3.3）。

为了使BCLK*、CCLK*信号满足多总线的规格要求，将9.8304MHz的时钟信号用S37电路驱动后再送给总线。如果这些信号从其他主控制模板提供时，就要去掉图中的跳线。

4.2.7　地址总线、数据总线缓冲器

图4.15是多总线的地址总线、数据总线缓冲器的逻辑电路图。在地址总线缓冲器中，8289的AEN/信号作为启动信号。在数据总线缓冲器中，通过A_0信号对交换缓冲器和通常的缓冲器进行转换之后，$\overline{AEN}$信号和$\overline{DEN}$信号相“与”作为启动信号（参考6.2.2）。

BHEN（高位字节允许）信号由$\overline{BHE}$（高位总线允许）信号和A_0信号产生。

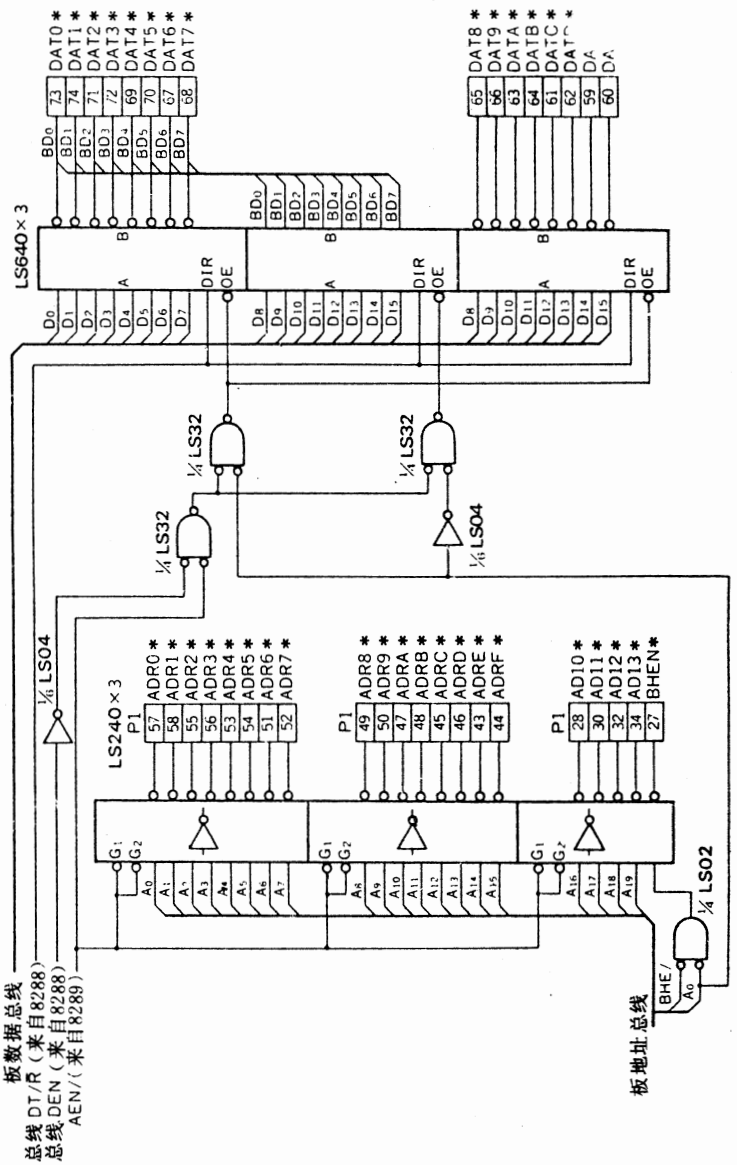


图4.15　多总线板的地址总线、数据总线缓冲器电路图

5 8086的汇编语言

在8086微处理器发表后不久，经常听到有人说："8086的汇编语言复杂难记"。8086的汇编语言果真那么复杂吗？回答是否定的。为了阅读程序简便起见，8086的汇编语言中引入了两个概念，即"有属性的符号"和"段"的概念。因此，8086微处理器有自己独特的伪指令和算符，乍一看程序似乎很复杂。但是，由于8086的汇编语言采用统一的助记符，一旦记住这些伪指令及算符等，就可以非常简单地描述了。本章将对这些伪指令和算符进行说明，并介绍一些简单的程序实例。

5.1 有属性的符号

在8085A的汇编语言中，符号全部当成数值处理。例如在下面的程序中，NUMBR、LOC、VAR同样都看成数值处理。

```
        ⋮
LOC:
NUMBR EQU $
VAR     DW 0
        ⋮
```

与8085A不同，8086的汇编语言中定义的符号具有属性。例如，在下面的程序中，NUMBR按数值属性处理，LOC按标号（在程序上的单元）属性处理，VAR按变量（数据）属性处理。

```
        ⋮
LOC:
NUMBR EQU OFFSET $
VAR     DW 0
        ⋮
```

而且，在8086的汇编语言中，区别立即数和变量不是通过助记符，而是通过属性进行的。例如：

```
MOV AX, NUMBR（立即数）
    VAR的偏移地址装入AX寄存器。
MOV AX, VAR（数据）
    VAR的内容装入AX寄存器。
MOV AX, LOC（错误）
    由于LOC既不是变量也不是数值，因此出错。
```

表5.1列出了符号的分类及其具有的属性。用汇编语言描述时，如果不充分理解定义的符号具有怎样的属性，就会产生意想不到的错误。在没有熟练之前，可首先把符号的英文字头记住，即N表示数值，V表示变量，L表示标号。

表5.1 符 号 的 分 类

分 类	内 容	属 性	定 义	参考指令
数 值	简单的数	无	由 EQU 指令定义	立即数的传送、运算指令
变 量	数 据	·单元 ·形式	由DB、DW、DD指令定义	访问存储器的传送、运算指令
标 号	单 元	单 元	由冒号“:”定义	转移、调用、循环指令

（1）数值

数值由EQU伪指令定义，只表示简单的数。

（2）变量

变量由DB、DW、DD伪指令定义，表示数据。具有单元（段和偏移）和类型（字节、字或双字）的属性。

在8086的汇编语言中，变量必须用上述伪指令定义。如下例所示，不能象8085A的汇编语言，使用“EQU伪指令定义地址及用数值指定直接地址”的方法。在访问EQU伪指令定义的符号时，全部作为立即数处理。

```
DATA    EQU     3
        ⋮
        LDA     DATA
        SHLD    DATA
        STA     3
```

（3）标号

标号由“:”定义，表示程序中的单元。标号通过转移指令、调用指令进行访问。它具有单元（段和偏移）的属性。

在INTEL公司的MDS汇编语言中，还有其他几种符号定义方法和属性，因过于复杂本书将不予多述。

5.2 段的指定

在8086的汇编语言中，将程序按段为单位进行分割，以便与段寄存器对应。因此，在程序上要进行段的指定，但是CP/M-86所属的汇编语言（简称CP/M汇编）和INTEL公司的MDS的汇编语言（简称MDS汇编）中，指定的方法差别是很大的。

程序例6、程序例7分别表示CP/M汇编和MDS汇编段的指定方法。CP/M汇编所采用的段的指定方法，只是8085A汇编语言的方法的扩充，它具有容易理解的优点，但是段的管理必须由用户进行。而MDS的汇编采用了新的方法，虽然最初难以理解，但由

于段的管理由汇编进行，所以这种方法不易产生错误。

CP/M汇编的目标程序是以绝对16进制形式输出的，因此不能与其他程序连接。MDS汇编的目标程序是以浮动形式输出的，因此可以与其他程序连接。而且它与其他程序连接之后，经过定位，可以再变成绝对16进制形式的目标程序。

5.2.1 CP/M汇编语言中段的指定

（1）段的指定

段的指定是通过CSEG、DSEG、ESEG、SSEG伪指令进行的。如果需要指定段地址，则该指定地址加在指令的后面，如程序例6的（A）、（B）所示。没有段结束的语句。各段的指定可以进行多次，但在汇编语言中只有CSEG、DSEG、ESEG、SSEG的区别，而没有其他的区别。

（2）对其他段的符号的访问

通过转移指令、调用指令向其他段分支转移时，可以采用JMPF、CALLF指令分支转移到任意的单元。但是，转移的目的段如果没有定义段地址，就按错误处理，如程序例6的(A)所示。

访问其他段的数据（变量）时，只描述符号。在CP/M汇编语言中不需要指定段修改，如果要访问的变量在CSEG、ESEG、SSEG所定义的段内，汇编语言就自动加上段修改前缀。

如程序例6的（A）所示，对于在两个数据（编码、附加码、堆栈）段中定义的变量，汇编语言不进行任何区别，只赋以偏移值。在这种情况下，用户必须先区别段，然后再编制程序。

5.2.3 MDS汇编中段的指定

（1）段的指定

段的指定通过SEGMENT伪指令进行。如程序例7的（A）、（B）所示，在SEGMENT伪指令之前必须加段名。在需要段地址时，就在AT运算符后写上地址。段的结束由ENDS伪指令指定。在MDS汇编语言中指定段时，不必区别段的类型（编码、数据、堆栈、附加码），而是在访问段时进行区别。

程序例 6　CP/M汇编中段的指定

```
                        (A) ----------------------------------------------
  F000                  段定义 CSEG   OF000H ; CODE SEGMENT 0
                               ORG    0000H
                        INIT:
0000 E91700      001A          JMP    MAIN        ┐向段内转移
0003 E82200      0028          CALL   MPROC       ┘
                        ;
0006 EA000000F1                JMPF   SPROC0      ┐向其它段转移
000B 9A000000F1                CALLF  SPROC1      ┘
                        ;
** ERROR NO: 23   MISSING SEGMENT INFORMATION IN OPERAND
0010 EA00000000                JMPF   SPROC2
                        ;          因ⓐ中没指定段地址故按错误处理
** ERROR NO: 23   MISSING SEGMENT INFORMATION IN OPERAND
0015 EA00000000                JMPF   SPROC3
                        ;
                        MAIN:
001A A10000                    MOV    AX,DODATA   ┐二变量分别在不同的
001D A11000                    MOV    AX,DIDATA   ┘数据段中，汇编不干予
                        ;
0020 26A12000                  MOV    AX,EDATA    ┐不需指定段修改前缀
                        ;                         ┘(由汇编自动指定)
0024 36A13000                  MOV    AX,SDATA
                        ;
  段修改前缀指令        MPROC:
0028 C3                        RET←——段内返回
                        ; (B) ----------------------------------------------
  F100                         CSEG   OF100H ; CODE SEGMENT 1
                        SPROC0:
                        ;
                        SPROC1:
0000 CB                        RETF←——段外返回
                        ; (C) ----------------------------------------------
                               CSEG   _______; CODE SEGMENT 2
                        ; (NOT DEFINE SEGMENT ADDRESS.)
                        SPROC2:
                        ;          ⓐ
                        SPROC3:
0000 CB                        RETF
                        ; (D) ----------------------------------------------
                               DSEG   ; DATA SEGMENT 0
                               ORG    0
0000 0000               DODATA DW     0
                        ; (E) ----------------------------------------------
                               DSEG   ; DATA SEGMENT 1
                               ORG    10H
0010 0000               DIDATA DW     0
                        ; (F) ----------------------------------------------
                               ESEG   ; EXTRA SEGMENT
                               ORG    20H
0020 0000               EDATA  DW     0
                        ; (G) ----------------------------------------------
                               SSEG   ; STACK SEGMENT
                               ORG    30H
0030 0000               SDATA  DW     0
                        ;      ----------------------------------------------
                        ;
                               END
```

（2）其他段的符号的访问

在MDS的汇编语言中，访问符号时，必须通过ASSUME伪指令，事前假定通过哪个寄存器进行访问。例如，在程序例7的(A)中，当访问符号时，要使用（也包括CSEGO本身）所有的段寄存器，所以采用ASSUME伪指令假定所有的段。例中的（D）、(E)、（F）、（G）由于不访问任何符号，所以不需要段的假定。

通过转移、调用指令向其他段转移时，可采用PTR算符产生FAR标号之后再转移。在不需要指定转移目的段的段地址时，就在定位时指定该地址。转移的标号除了程序例7的（C）中通常的程序标号外，还可以按同例的（D）中的描述。

在访问其他段的数据（变量）时，数据段内的变量只有符号本身，除此之外的段的变量则用段修改前缀作为指定的符号加以描述。如果不进行段修改前缀的指定，就会产生程序例7的（A）的错误（被访问的符号在程序之后）。如果要访问的符号，没有在ASSUME伪指令假定的段内时，也会产生错误。这时，如例中下面所示，如果再重新用ASSUME伪指令假定，就可以消除错误。当然在此之前还应该变更DS寄存器（例中已省略）。

用冒号":"定义标号时，必须事先通过ASSUME伪指令假定代码段。如果不进行这个假定，上述的定义将全部当错误处理。

用CP/M的汇编语言的CSEG、DSEG、SSEG、ESEG伪指令和MDS的汇编语言的ASSUME伪指令对段的指定，只不过为了通知汇编程序在进行汇编时，生成什么样的编码。汇编语言本身在执行汇编的程序时，对段寄存器中装入什么值是一概不清楚的。

如何向段寄存器中装入正确的段地址，完全是用户的责任，因此在汇编的程序中，应该有这方面的指令。段地址通常装入汇编语言中假定的值，但有时必须装入完全无关的值。

程序例7 MDS汇编语言中段的指定

```
                                1   ;          段定义          段地址指定
                                2   ; (A) ------------------------------------------
----                            3   CSEG0 SEGMENT AT 0F000H ; CODE SEGMENT 0
                                4   ;         假定该段中的段寄存器
                                5     ASSUME CS:CSEG0,DS:DSEG0,ES:ESEG,SS:SSEG
0000                            6         ORG    0000H
0000                            7   INIT:
0000 EB1890                     8         JMP    MAIN     ] 向段内转移
0003 E82B00                     9         CALL   MPROC    ]
                               10   ;
0006 EA000000F1                11         JMP    FAR PTR SPROC0 ] 向段外转移
000B 9A000000F1                12         CALL   FAR PTR SPROC1 ]   由PTR算符产
                               13   ;                             ]   生FAR标号
0010 EA0000----      R         14         JMP    FAR PTR SPROC2 ]
0015 9A0000----      R         15         CALL   FAR PTR SPROC3 ]
                               16   ;
001A                           17   MAIN:                  因为ASSUME假定的段中
001A A10000                    18         MOV    AX,D0DATA  没有，所以作错误处理
001D 909090                    19         MOV    AX,D1DATA
*** ERROR #5, LINE #19, (PASS 2) OPERAND NOT REACHABLE FROM SEGMENT REGISTERS
                               20   ;
                               21     ASSUME DS:DSEG1       ] ASSUME中重新假定通过
0020 A10000                    22         MOV    AX,D1DATA  ]
                               23   ;
0023 26A12000                  24         MOV    AX,ES:EDATA   需指定段修改前缀
0027 26A190                    25         MOV    AX,EDATA
*** ERROR #3, LINE #25, (PASS 2) INSTRUCTION SIZE BIGGER THAN PASS 1 ESTIMATE
                               26   ;
002A 36A13000                  27         MOV    AX,SS:SDATA
002E 36A190                    28         MOV    AX,SDATA
*** ERROR #3, LINE #28, (PASS 2) INSTRUCTION SIZE BIGGER THAN PASS 1 ESTIMATE
                               29   ;
0031                           30   MPROC:
0031 C3                        31         RET
                               32   ;
----                           33   CSEG0 ENDS ←—— 段结束
                               34   ;
```

```
                                35      ; (B) ---------------------------------------
----                            36      CSEG1 SEGMENT AT 0F100H ; CODE SEGMENT 1
                                37      ;
0000                            38      SPROC0  LABEL FAR
                                39      ;
0000                            40      SPROC1  PROC  FAR
0000 CAFEFF                     41              RET   FAR
                                42      SPROC1 ENDP
                                43      ;
----                            44      CSEG1 ENDS
                                45      ;
                                        ;       ---------------------------------------
                                46      ; (C) ---------------------------------------
----                            47      CSEG2 SEGMENT           ; CODE SEGMENT 2
                                48      ; (NOT DEFINE SEGMENT ADDRESS.)
                                49      ;
                                50          ASSUME CS:CSEG2
0000                            51      SPROC2:
                                52      ;
0000                            53      SPROC3:
0000 C2FEFF                     54              RET   FAR
                                55      ;
----                            56      CSEG2 ENDS
                                57      ;
                                58      ; (D) ---------------------------------------
----                            59      DSEG0 SEGMENT ; DATA SEGMENT 0
                                60      ;
0000                            61              ORG   0
0000 0000                       62      D0DATA  DW    0
                                63      ;
----                            64      DSEG0 ENDS
                                65      ;
                                66      ; (E) ---------------------------------------
----                            67      DSEG1 SEGMENT ; DATA SEGMENT 1
                                68      ;
0000                            69              ORG   0
0000 1000                       70      D1DATA  DW    10H
```

```
                                71      ;
----                            72      DSEG1 ENDS
                                73      ;
                                74      ; (F) ----------------------------------------
----                            75      ESEG  SEGMENT ; EXTRA SEGMENT
                                76      ;
0020                            77              ORG   20H
0020 0000                       78      EDATA   DW    0
                                79      ;
----                            80      ESEG  ENDS
                                81      ;
                                82      ; (G) ----------------------------------------
----                            83      SSEG  SEGMENT
                                84      ;
0030                            85              ORG   30H
0030 0000                       86      SDATA   DW    0
                                87      ;
----                            88      SSEG  ENDS
                                89      ;
                                90      ;     ----------------------------------------
                                91      ;
                                92              END
```

5.3 伪指令和算符

5.3.1 伪指令

表5.2中是经常使用的伪指令。EQU、ORG、DB、DW 伪指令与8085A的汇编语言相同，但是它所定义的符号具有属性。

表5.2 经常使用的伪指令

功　能	CP/M 的汇编语言	MDS的汇编语言
数值的定义	EQU	EQU
偏移值的指定	ORG	ORG
段的定义	CSEG, DSEG	SEGMENT
	ESEG, SSEG	ENDS
段的假定	无	ASSUME
字节变量的定义	DB	DB
字变量的定义	DW	DW
双字变量的定义	DD	DD
字节数组的定义	RB或RS	无
字数组的定义	RW	无

（注）RS定义的数组不具有属性

关于段的指定方法在前节已经叙述过了。关于数组的定义，在CP/M汇编语言中采用RB、RS、RW 伪指令定义。这些伪指令不能对定义的内容初始化。另外，RS定义的符号不具有属性，这一点务必注意。在MDS的汇编语言中没有定义数组的伪指令。可以通过DUP重复DB、DW的指定。

DD伪指令用于定义双字（32位）的变量（参见程序例8的(I)）。

5.3.2 算符

8086的汇编语言中仍能采用8085A中采用的AND、OR等算符。

但是除了具有属性的符号外，下面要说明的算符及其描述方法也是很重要的，必须很好理解。

（1）OFFSET、SEG

这两个算符对变量和标号起作用，分别产生该单元的偏移地址和段地址。如程序例8的（A）所示，在最前面的MOV指令中，由于DWDATA是作为变量定义的，所以它表示存储器的内容。下面的两条指令中，通过OFFSET和SEG算符分别产生偏移地址和段地址，这些地址是以立即数据装入的。

（2）PTR

PTR是二项算符，左边一项用BYTE、WORD、DWORD之一描述，右边一项描述指定的符号或寻址方式，其意义是按左项的类型生成右项。

程序例8的（B）是改变变量类型的例子。如例中所示，如果把字变量装入字节寄存器，或者把字节变量装入字寄存器，都会产生错误，为此，需通过PTR算符重新生成变量的类型。

程序例8的（C）是指定间接地址时，使用PTR算符的例子。向间接地址指定的存储器中存入立即数时，并不知道是以字节数据存入的、还是以字数据存入的。因此，需要采用PTR算符进行指定。

程序例8的（D）表示存储器间接转移的例子。PTR算符的用法与（B）相同。

（3）其他

程序例8的（D）表示用数值指定偏移地址的方法。在CP/M的汇编语言中，可以采用“·”指定偏移地址。

程序例8的（F）表示指定段修改前缀的方法。如例所示，段修改前缀由冒号“:”进行指定。

在CP/M的汇编语言中，对变量不需要指定段修改前缀，而由汇编程序自动指定。在MDS的汇编语言中，对前面已经定义的变量同样不需要指定，但在后面定义的变量就必须指定。

```
                                CSEG  ; CODE SEGMENT 0
                                ORG   0000H
                        ; (A) ----------------------------------------------
                        ;     使用 OFFSET · SEG 算符的例子
0000 A10002                     MOV   AX,DWDATA
0003 B80002                     MOV   AX,OFFSET DWDATA
0006 B80001                     MOV   AX,SEG DWDATA
                        ; (B) ----------------------------------------------
                        ;     使用 PTR 算符的例 1
0009 A10002                     MOV   AX,DWDATA   ] 通常的方法
000C A00202                     MOV   AL,DBDATA   ]
                        ;
** ERROR NO:  7   OPERAND(S) MISMATCH INSTRUCTION ]
000F 909090909090               MOV   AL,DWDATA   ] 由于变量与装入寄存器
** ERROR NO:  7   OPERAND(S) MISMATCH INSTRUCTION ] 类型不同出错
0015 909090909090               MOV   AX,DBDATA   ]
                        ;
001B A00002                     MOV   AL,BYTE PTR DWDATA ] 重新生成变量类型通过
001E A10202                     MOV   AX,WORD PTR DBDATA ]
                        ; (C) ----------------------------------------------
                        ;     使用 PTR 算符例 2
0021 C606020201                 MOV   DBDATA,1    向存储器存入立即数据
                        ;
** ERROR NO: 21   MISSING TYPE INFORMATION IN OPERAND(S)
0026 909090909090               MOV   [BX],1      由于未指定类型出错
                        ;                         (不知道存入存储器的字中还是字节中)
002C C60701                     MOV   BYTE PTR [BX],1  指定类型通过
                        ;
                        ; (D) ----------------------------------------------
                        ;     使用 PTR 算符例 3
002F E85100       0083          CALL  MPROCO ] 直接（相对）地址
0032 E94E00       0083          JMP   MPROCO ]
0035 FF260B02                   JMP   WPOINT   存储器间接转移
                        ;
** ERROR NO:  7   OPERAND(S) MISMATCH INSTRUCTION ] 由于 BPOINT 是字节变量出错
0039 9090909090                 JMP   BPOINT     ]
                        ;
003E FF260702                   JMP   WORD PTR BPOINT 生成字类型通过
```

```
0042 EA000000F2                 JMPF   SPROC0   ┐直接地址
0047 9A000000F2                 CALLF  SPROC0   ┘
004C FF2E0D02                   JMPF   DPOINT   存储器间接转移
                        ;
** ERROR NO:  7   OPERAND(S) MISMATCH INSTRUCTION ┐由于WPOINT是字变量，出错
0050 9090909090                 JMPF   WPOINT   ┘
                        ;
0055 FF2E0B02                   JMPF   DWORD PTR WPOINT 生成双字类型通过
                        ; (E) ------------------------------------------
                        ;       数值寻址（只适用CP/M）
0059 B003                       MOV    AL,3     立即数据
                        ;
005B A00300                     MOV    AL,.3    地址（存储器内容）
                        ; ----------------------------------------------
                        ; (F) ------------------------------------------
                        ;       指定段修改前缀
005E A10002                     MOV    AX,DWDATA
0061 2EA18300                   MOV    AX,CS:CDATA ┐
0065 26A11000                   MOV    AX,ES:EDATA ┤通常的指定方法
0069 36A12000                   MOV    AX,SS:SDATA ┘
                        ;
006D 2EA18300                   MOV    AX,CDATA  实际上由汇编自动加段修改前缀
          汇编错误?     ;
0071 8B07  (执行结果相同)       MOV    AX,[BX]
0073 2E8B870000                 MOV    AX,CS:[BX] ┐
                        ;                         ┤间接地址需要指定段修改前缀
0078 8B4600                     MOV    AX,[BP]    ┤
007B 3E8B4600                   MOV    AX,DS:[BP] ┘
                        ;
007F 26C60703                   MOV    ES:BYTE PTR [BX],3 与PTR算符混用例子
                        ; (G) ------------------------------------------
                        MPROC:
0083 0000               CDATA   DW     0
                        ; (H) ------------------------------------------
  F200                          CSEG   0F200H ; CODE SEGMENT 1
                        SPROC0:
0000 CB                         RETF
                        ; (I) ------------------------------------------
```

```
  0100                          DSEG  100H : DATA SEGMENT
                                ORG   200H
 0200 0000              DWDATA  DW    0
 0202 00                DBDATA  DB    0

** ERROR NO: 20   ILLEGAL EXPRESSION ELEMENT
 0203 00000000          DDDATA  DD    0
                        ;
 0207 00                BPOINT  DB    0
 0208 00                        DB    0
                        ;
** ERROR NO: 10 **  NEAR: "MPROC"   UNDEFINED ELEMENT OF EXPRESSION
 0209 0000              WPOIN   DW    MPROC
                        ;
 020B 8300              WPOINT  DW    OFFSET MPROC0
                        ;
 020D 000000F2          DPOINT  DD    SPROC0
                        ;
                        ; (J) ------------------------------------------------
                                ESEG  : EXTRA SEGMENT
                                ORG   10H
 0010 0000              EDATA   DW    0
                        ; (K) ------------------------------------------------
                                SSEG  : STACK SEGMENT
                                ORG   20H
 0020 0000              SDATA   DW    0
                        ;       ------------------------------------------------
```

在CP/M汇编中不能定义32位的数据（汇编的错误？）

只定义偏移地址时需要OFFSET算符

标号的段偏移地址由DD伪指令定义

5.4 间接地址指定和字符串处理指令、XLAT指令的描述方法

5.4.1 间接地址指定

间接地址的指定采用在括号〔 〕内写上地址寄存器的方法。CP/M的汇编语言和MDS的汇编语言对间接地址指定的描述方式略有不同。

程序例9是CP/M汇编语言中的描述方式，程序例10是MDS汇编语言中的描述方式。如例中所示，虽然二者同是寄存器间接寻址方式，但是在基址+变址或有位移的寻址方式时，描述方式稍有差别，为了使在任何一种汇编语言中都不出错，可按下面的统一描述。

（1）基址+变址方式的描述方法是两个寄存器相加，外面用〔 〕括起来。基址寄存器和变址寄存器哪一个写在前面都可以，与它们的先后顺序无关。

〔例〕MOV AX, 〔SI+BP〕

（2）把变量的偏移地址作为位移描述时，应在括〔 〕前写上符号。

〔例〕MOV AX, DATA〔BX〕

（3）把数值作为位描述时，应在括〔 〕的前面或后面加上加号“+”。

〔例〕MOV AX, 〔DI+BP〕+3

MOV AX, NUM+〔SI〕

（4）要避免使用带变量的偏移地址和数值的位移。

5.4.2 字符串处理指令和XLAT指令的描述方法

本来字符串处理指令和XLAT指令都是无操作数的指令。但是，由于二者都是存储器存取指令。所以，在字符串处理指令中需要指定有无段修改前缀，并指定是字节还是字；而XLAT指令中，需要指定有无段修改前缀。为此采用了隐含操作数的方法。

程序例9　CP/M汇编语言间接地址指定

```
        CSEG
OFST    EQU   3
; (A) ----------------------------------------
;       寄存器间接及基址+变址
        oMOV   AX,[BX]
        oMOV   AX,[SI+BX]
        oMOV   AX,[BX+SI]
; (B) ----------------------------------------
;       变量的偏移地址为位移
        MOV    AX,.3[BX]
        MOV    AX,OFST[BX]          由于OFST是数值,
        oMOV   AX,DATA[BX]          本应按错误处理
; (C) ----------------------------------------
;       数值为位移
        MOV    AX,[BX]+3            ┐
        MOV    AX,[BX+SI]+OFST      ├ 加在后面
        MOV    AX,[BX]+OFFSET DATA  ┘
;
        MOV    AX,3+[SI+BX]         ┐
        MOV    AX,OFST+[BX]         ┘ 加在前面
;
; (ERROR) ****************************************
;       错误处理举例
        MOV    AX,[SI][BX]          ┐
        MOV    AX,[SI]+[BX]         ├ 只有MDS汇编
;                                   │ 用这种描述
        MOV    AX,[SI+BX+3]         │
        MOV    AX,[BX+OFST]         ┘
        MOV    AX,[BX+OFFSET DATA]
;
        MOV    AX,OFFSET DATA[BX]
        MOV    AX,OFST[BX]+OFFSET DATA
;
        MOV    AX,.3[BX]+OFST
        MOV    AX,.3[SI+BX]+OFFSET DATA
        MOV    AX,OFST[BX+SI]+3
        MOV    AX,DATA[BX]+OFST
        MOV    AX,DATA[SI+BX]+3
;
;*********************************************
;
        DSEG
        ORG    OFST
DATA    DW     0
;
```

程序例11是采用上述方法的实例。对字符串处理指令来说，隐含操作数采用了AX、AL寄存器(只能用于CP/M的汇编语言)及隐含变量。在XLAT指令中采用了隐含变量。这些隐含变量只要符合定义的段和类型，什么内容也没关系。实际的偏移地址采用SI、DI寄存器或BX寄存器的内容。在需要段修改前缀时，与通常的地址描述方法一样，用冒号“:”指定。操作数CS:AX在物理上完全没有意义，它只不过是为了指明在汇编中要进行CS寄存器的段修改前缀以

程序例10　MDS汇编语言中间接地址的指定

●表示在CP/M汇编中也可采用

```
;
CSEG   SEGMENT
       ASSUME  DS:DSEG
OFST     EQU   3
; (A)  ----------------------------------------
;      寄存器间接及基址+变址
         ●MOV    AX,[BX]
         ●MOV    AX,[SI+BX]
         ●MOV    AX,[BX+SI]
;
           MOV   AX,[SI][BX]   ┐只在MDS汇编中
           MOV   AX,[SI]+[BX]  ┘使用该描述
; (B)  ----------------------------------------
;      变量的偏移地址为位移
           MOV   AX,DATA[BX]
; (C)  ----------------------------------------
;      数值为位移
         ●MOV    AX,[BX]+3                  ┐
         ●MOV    AX,[BX+SI]+OFST            ├加在后面
         ●MOV    AX,[BX]+OFFSET DATA        ┘
;
         ●MOV    AX,3+[SI+BX]               ┐加在前面
         ●MOV    AX,OFST+[BX]               ┘
;
           MOV   AX,[SI+BX+3]  ┐只在MDS汇编中
           MOV   AX,[BX+OFST]  ┘使用该描述
; (D)  ----------------------------------------
;      变量的偏移地址+数值的位移
           MOV   AX,DATA[BX]+3
           MOV   AX,DATA[SI+BX]+OFST
;
; (ERROR) ****************************************
;      错误处理的举例
           MOV   AX,OFST[BX]          ←——OFST不是变量
           MOV   AX,OFFSET DATA+[BX]
           MOV   AX,[BX]+DATA         ┐
           MOV   AX,DATA+[BX]         ├DATA不是数值
           MOV   AX,[BX+OFFSET DATA]  ┘
;
;****************************************************
;
CSEG   ENDS
;
DSEG   SEGMENT
         ORG   3
DATA     DW    0
;
         END
```

及处理要以字为单位进行而已。

CMPS指令与其他指令刚好相反，源操作数写在左边，而目标操作数写在右边，使用时要注意。

程序例11　字符串处理指令、XLAT指令的描述举例

```
                                CSEG
                                ORG     0
  0010                  NARRAY  EQU     10H
                        ; (A) ------------------------------------------
                        ;       字符串指令的基本形
0000 BE0000                     MOV     SI,OFFSET DARRAY
0003 BF0000                     MOV     DI,OFFSET EARRAY
0006 B90800                     MOV     CX,NARRAY/2
0009 FC                         CLD
000A F3A5                   REP MOVS    AX,AX
                        ; (B) ------------------------------------------
                        ;       段修改前缀                    只在CP/M汇编中
000C F32EA5                 REP MOVS    AX,CS:AX              可用该描述方法
000F F326A5                 REP MOVS    AX,ES:AX
0012 F336A5                 REP MOVS    AX,SS:AX
                        ;
0015 F3A4                   REP MOVS    EARRAY,DARRAY         通常的
0017 F32EA4                 REP MOVS    EARRAY,CS:CARRAY      描述方法
001A F326A4                 REP MOVS    EARRAY,ES:EARRAY
001D F336A4                 REP MOVS    EARRAY,SS:SARRAY
                        ;
0020 F32EA4                 REP MOVS    EARRAY,CARRAY
                        ;               实际上不用指定段修改前缀
0023 F0F32EA4           LOCK REP MOVS   EARRAY,CARRAY
                        ;               与其它前置指令混用的例子
** ERROR NO:  7   OPERAND(S) MISMATCH INSTRUCTION
0027 F39090909090           REP MOVS    CS:CARRAY,DARRAY
     90
                        ;
                        ; (C) ------------------------------------------
                        ;       XLAT指令的描述例
002E D7                         XLAT    DARRAY
002F 2ED7                       XLAT    CS:CARRAY             段修改前缀
0031 2ED7                       XLAT    CARRAY
                        ;
** ERROR NO:  7   OPERAND(S) MISMATCH INSTRUCTION
0033 9090                       XLAT
                        ;
                        ; (D) ------------------------------------------
                                DSEG
0000                    DARRAY  RB      NARRAY
                        ;
                                ESEG
0000                    EARRAY  RB      NARRAY
                        ;
                                SSEG
0000                    SARRAY  RB      NARRAY
                        ;
                                CSEG
0000                    CARRAY  RB      NARRAY
```

闭锁　　段修改前缀　　重复前置指令

5.5 用汇编语言写的程序实例

5.5.1 单步中断的程序

程序例12为采用单步中断的监控程序的例子。在用戶程序（进行调试时的程序）上，执行一条指令后，产生单步中断，分支转移到该监控程序。

例中（A）的部分将中断处理程序的偏移和段地址设置到4号地址。（因为单步中断的中断类型为1，所以是1×4=4号地址）。

例中（B）的部分是中断处理程序。通过STORE—REG子程序把用戶程序中的寄存器的内容全部保存在存储器中以后，执行监控程序。（在该例中已省略）。从监控程序向用戶程序的分支转移，是在SBRANCH—USER程序把保存的寄存器的内容返回各寄存器之后再进行的。

例中（C）的部分是保存寄存器内容的程序。在8086微处理器中，由于可以在所有的寄存器和存储器间进行传送操作，所以很容易进行寄存器内容的保存。寄存器的内容保存以后，首先将子程序的返回地址送入AX寄存器，接着保存指令指针、CS寄存器、标志的内容。然后再保存堆栈指针，改变成监控程序的堆栈。最后将返回地址压入监控程序的堆栈后返回。

例中（D）是设置陷阱标志后分支转移到用戶程序的程序。首先将保存的寄存器内容全部返回各寄存器。然后设置保存标志的陷阱标志位，并按标志、CS寄存器、指令指针的顺序进行进栈操作，执行IRET指令。由于在8086的标志中装的是陷阱标志所设置的内容，因此返回后再执行一条指令，会再次发生单步中断，又返回到SINGLE-INT上。

程序例12　单步中断程序举例

```
  2000                    CSEG   2000H
                    ;
                    ; (A) ++++++++++++++++++++++++++++++++++++++
                    ;                                  在4号地址设置单步中断
0000 B80000                 MOV    AX,0                处理程序的地址
0003 8ED8                   MOV    DS,AX
0005 BB0400                 MOV    BX,1*4
                    ;
0008 C7071100               MOV    WORD PTR [BX],OFFSET SINGLE_INT
000C C747020020             MOV    WORD PTR [BX]+2,SEG SINGLE_INT
                    ;
                    ; (B) ++++++++++++++++++++++++++++++++++++++
                    ;                  将中断产生前的寄存器内容（调试中程序
                    SINGLE_INT:        的寄存器内容）保留到存储器中
0011 FA                     CLI
0012 E80300  0018           CALL   STORE_REG
                    ;  ┌ 监控程序 ┐
                    ;  │          │
                    ;  │进入接受上述寄存器内容显示及新的监视命令的程序
                    ;  └          ┘
0015 E95000  0068           JMP    S_BRANCH_USER
                    ;寄存器的内容复原，转到调试中的程序
                    ;++++++++++++++++++++++++++++++++++++++++++++
                    ; (C) ++++++++++++++++++++++++++++++++++++++
                    ;
                    STORE_REG:
0018 1E                     PUSH   DS                  设置监控的数据段地址
0019 2E8E1E6200             MOV    DS,CS:MONI_DS
001E A30000                 MOV    USER_AX,AX   ┐
0021 891E0200               MOV    USER_BX,BX   │
0025 890E0400               MOV    USER_CX,CX   │ 将调试中程序的寄存器
0029 89160600               MOV    USER_DX,DX   │ 内容保留到存储器中
002D 89360800               MOV    USER_SI,SI   │
0031 893E0A00               MOV    USER_DI,DI   │
0035 892E0C00               MOV    USER_BP,BP   │
                    ;                           │
0039 8C161200               MOV    USER_SS,SS   │ 8086中堆栈的内容可送入存
003D 8C061400               MOV    USER_ES,ES   │ 储器，也可从存储器取出
0041 8F061000               POP    USER_DS      ┘
                    ;
0045 58                     POP    AX ←— 该程序的返回地址
                    ;
0046 8F061A00               POP    USER_IP      ┐ 保留指令指针、
004A 8F061C00               POP    USER_CS      │ CS寄存器标志
004E 8F061E00               POP    USER_FL      ┘
                    ;
0052 89260800               MOV    USER_SP,SP ←— 保留堆栈指针
                    ;                           （产生中断前的值）
0056 2E8E166600             MOV    SS,CS:MONI_SS ┐ 监控堆栈置位
005B 2E8B266400             MOV    SP,CS:MONI_SP ┘
                    ;
0060 50                     PUSH   AX ←— 返回地址进入监控堆栈区
                    ;
0061 C3                     RET
```

```
                        ;
0062 0028               MONI_DS DW      SEG USER_AX
0064 1C02               MONI_SP DW      OFFSET MONI_SP_TOP
0066 0028               MONI_SS DW      SEG MONI_SP_TOP
                        ;
                        ;+++++++++++++++++++++++++++++++++++++++++++
                        ; (D) ++++++++++++++++++++++++++++++++++++++
                        ;
                        S_BRANCH_USER:
0068 A10000                     MOV     AX,USER_AX  ┐
006B 8B1E0200                   MOV     BX,USER_BX  │
006F 8B0E0400                   MOV     CX,USER_CX  │
0073 8B160600                   MOV     DX,USER_DX  │
0077 8B360800                   MOV     SI,USER_SI  │ 调试中程序的寄存器内容
007B 8B3E0A00                   MOV     DI,USER_DI  │
007F 8B2E0C00                   MOV     BP,USER_BP  │
0083 8E061400                   MOV     ES,USER_ES  ┘
                        ;
0087 8E161200                   MOV     SS,USER_SS  ┐ 将堆栈设置在调试
008B 8B260E00                   MOV     SP,USER_SP  ┘ 程序的堆栈中
                        ;
008F 810E18000001               OR      USER_FL,0100H ←── 设置陷阱标志
0095 FF361800                   PUSH    USER_FL     ┐
                        ;                           │ 标志、CS寄存器指令指针进堆
0099 FF361600                   PUSH    USER_CS     │
009D FF361A00                   PUSH    USER_IP     ┘
                        ;
00A1 8E1E1000                   MOV     DS,USER_DS ←── 数据段的内容
                        ;
00A5 CF                         IRET
                        ;
                        ;+++++++++++++++++++++++++++++++++++++++++++
                          由于设置了陷阱标志，所以在返回后执行单个
                          指令，将再次产生单步中断
                        ; (E) ++++++++++++++++++++++++++++++++++++++
                        ;
  2800                          DSEG    2800H
                        ;
0000 0000               USER_AX DW      0       ┐
0002 0000               USER_BX DW      0       │
0004 0000               USER_CX DW      0       │
0006 0000               USER_DX DW      0       │
0008 0000               USER_SI DW      0       │
000A 0000               USER_DI DW      0       │
000C 0000               USER_BP DW      0       │ 调试程序的寄存器保留区
000E 0000               USER_SP DW      0       │
0010 0000               USER_DS DW      0       │
0012 0000               USER_SS DW      0       │
0014 0000               USER_ES DW      0       │
0016 0000               USER_CS DW      0       │
0018 0000               USER_FL DW      0       │
001A 0000               USER_IP DW      0       ┘
                        ;
001C                            RW      100H    ┐ 监控使用的堆栈区
                        ;                       │
                        MONI_SP_TOP:            ┘
                        ;
                        ;+++++++++++++++++++++++++++++++++++++++++++
                        ;
                                END
```

5.5.2 其他程序举例

程序例13是把8086作为控制台和打印机专用的从属处理器工作时的程序例子。虽然在初始化程序中也进行TTY控制器的初始化，但在主程序中不进行TTY的控制。图5.1为程序例13的流程图。

程序例13中，在I/O控制器和存储器初始化之后，对控制台和打印机的输入输出进行检查，并进行必要的处理。向打印机输出数据是通过软件组成的先进先出（FIFO）缓冲区进行的，以便减少主CPU的等待时间。

程序例13　8086汇编语言的程序举例

```
                ;************************************************
                ;       ------------------------------------
                ;        PROGRAM EXAMPLE OF 8086 ASSEMBLER
                ;       ------------------------------------
                ;
                ;  List of example procedure
                ;    (1) Initialise of 8251A
                ;    (2) Initialise of 8259A
                ;    (3) Control of 8251A
                ;    (4) Test and set
                ;    (5) First in first out RAM constructed -
                ;         by software
                ;************************************************
                ;
                ;++++++++++++++++++++++++++++++++++++++++++++++++
                ; Address and constaint define
                ;++++++++++++++++++++++++++++++++++++++++++++++++
                ; Code segment & offset
F000            CODE_SEG    EQU   0F000H
0000            CODE_OFST   EQU   00000H
0300            IO_TABLE    EQU   CODE_OFST+300H
0420            RAM_TABLE   EQU   CODE_OFST+420H
                ; Interrupt vector offset
0098            INT_OFST    EQU   00020H+(6*4)
                ; Data  segment & offset
0100            DATA_SEG    EQU   00100H
0000            DATA_OFST   EQU   00000H
                ; Linprinter buffer area
0200            L_BUF_SEG   EQU   00200H ; Segment ┐
0000            L_BUF_OFST  EQU   00000H ; Offset  ├ 打印机的FIFO缓
1000            L_LP_BUF    EQU   1000H  ; Length  ┘ 冲区
                ; Length of stack
0200            L_STACK     EQU   200H
                ;------------------------------------------------
                ; Interrupt controller address (8259A)
00C0            INTCW0      EQU   0C0H       ; Command word 0
00C2            INTCW1      EQU   INTCW0+2   ; Command word 1
                ;
```

```
                ; TTY and console controller address (8251A)
00D9            TTY_BASE        EQU     0D9H            ; TTY
00D8            CON_BASE        EQU     0D8H            ; Console
                ;
                ; 8259A initialise code.
0013            ICW1            EQU     013H            ; ICW1 of 8259A
                ;  ( Edge sense, Single )
0020            ICW2            EQU     020H            ; ICW2 of 8259A
                ;  ( Type vector -- Int. type is 32 to 39 )
000D            ICW4            EQU     00DH            ; ICW4 of 8259A
                ;  ( Fully nested mode, Buffer master mode,
                    normal EOI)
003F            OCW1            EQU     03FH            ; OCW1 Of 8259A
                ;  ( Interrupt mask -- Enable INT6 and INT7 )
0020            EOI             EQU     020H            ; EOI code (OCW2)
                  ;  ( Non spesific EOI )
                  ;
  0000            LP_STS        EQU   0               ; Dummy define.
                  ;+++++++++++++++++++++++++++++++++++++++++++++++++
                  ; Code area
                  ;+++++++++++++++++++++++++++++++++++++++++++++++++
                  ;
  F000                          CSEG    CODE_SEG
                                ORG     CODE_OFST
                ;
0000 FA                 CLI     ; Disable interrupt.
0001 FC                 CLD     ; Clear direction flag.
                ;
0002 B80002             MOV     AX,SEG STACK_TOP    ; Set stack -
0005 8ED0               MOV     SS,AX               ; pointer.
0007 BC0210             MOV     SP,OFFSET STACK_TOP
                ;
000A E8AA00 00B7        CALL    INITIO ; Initialise i/o controller.
                ;
                ; Initialise registers and memory.
000D 2EA12004           MOV     AX,CS:SEG_SLAVE ; Set data segment
0011 8ED8               MOV     DS,AX                reg.
                ;
0013 8EC0               MOV     ES,AX
0015 B000               MOV     AL,0                ; Clear critical -
0017 B90800             MOV     CX,L_CRTBUF         ; Buffer.
001A BF0000             MOV     DI,OFFSET SLV_FLAG
001D F3AA             REP STOS  AL
                ;
                ; Initialise lineprinter buffer pointer.
001F B80002             MOV     AX,L_BUF_SEG ; Segment.
0022 8EC0               MOV     ES,AX
                CLRPRBUF:
0024 BE0000             MOV     SI,B_LP_BUF  ; Source pointer.
0027 8BFE               MOV     DI,SI        ; Output pointer.
                ;
                ;-----------------------------------------
                ; Console input check.
                CCONIN:
0029 FB                 STI
002A 2E8B160803         MOV     DX,CS:CON_CMND
002F EC                 IN      AL,DX       ; RX-ready check.
0030 2402               AND     AL,002H
```

常用例子

```
0032 740D   0041        JZ      CCONOU      ; If no input jump.
                    ; Input data and set to buffer.
0034 B4FF               MOV     AH,0FFH     ; Set flag.
0036 2E8B160603         MOV     DX,CS:CON_DATA
003B EC                 IN      AL,DX       ; Get data.
003C F087060100     LOCK XCHG   B_CO_IN,AX  ; Set data and flag.
                    ;
                    ; Console output check.
                    CCONOU:
0041 2E8B160803         MOV     DX,CS:CON_CMND
0046 EC                 IN      AL,DX       ; Out. buf. is empty?
0047 2481               AND     AL,081H     ;  (TX-ready & DSR = 1 ?)
0049 3C81               CMP     AL,081H     ; If not jump.
004B 7511   005E        JNZ     CLPOUT
                    ;
004D B400               MOV     AH,0        ; Reset flag.
004F F087060300     LOCK XCHG   AX,B_CO_OUT  ; Check slave CPU -
0054 84E4               TEST    AH,AH       ; request output.
0056 7406   005E        JZ      CLPOUT      ; If no request jump.
                    ; Output data
0058 2E8B160603         MOV     DX,CS:CON_DATA
005D EE                 OUT     DX,AL
                    ;
                    ; Check Lineprinter busy and if not -
                    ; output one byte data.
                    CLPOUT:
005E E400               IN      AL,LP_STS ; Check printer input -
0060 24FF               AND     AL,0FFH   ; busy and if so jump.
                    ; (This check is dummy.)                    伪检查
0062 741C   0080        JZ      CLPDAT
                    ;
0064 3BFE               CMP     DI,SI       ; Is data buffer empty ?
0066 7318   0080        JAE     CLPDAT      ; If so jump.
0068 268A05             MOV     AL,ES:[DI]  ; Get data from buffer -
006B E84800 00B6        CALL    LPOUT       ; and output.
006E BD0200             MOV     BP,2        ; Set timer.
0071 47                 INC     DI          ; Increment pointer And-
0072 81FF0010           CMP     DI,T_LP_BUF ; if it is top of buffer-
0076 7208   0080        JB      CLPDAT      ; reset to bottom -
0078 81EF0010           SUB     DI,L_LP_BUF ; and update source -
007C 81EE0010           SUB     SI,L_LP_BUF ; pointer by same value.
                    ;
                    ; Lineprinter output check.
                    ;  Check slave CPU output request.
0080 8BDF           CLPDAT:
0082 81C30010           MOV     BX,DI       ; Is data buffer is full ?
0086 3BF3               ADD     BX,L_LP_BUF
0088 731B               CMP     SI,BX
             00A5       JAE     CCLRBUF     ; If so jump.
008A B400           ;
008C F087060500         MOV     AH,0        ; Reset flag.
0091 84E4           LOCK XCHG   AX,B_LP_OUT ; Check slave CPU -
0093 7410               TEST    AH,AH       ; request output.
           00A5         JZ      CCLRBUF     ; If no request jump.
0095 8BDE           ;
0097 81FE0010           MOV     BX,SI
                        CMP     SI,T_LP_BUF ; If data pointer is not -
```

```
009B 7204    00A1          JB     SAVEDAT      ; overflow top of buffer -
009D 81EB0010              SUB    BX,L_LP_BUF  ; move pointer again.
                        SAVEDAT:
00A1 268807                MOV    ES:[BX],AL   ; Save data and -
00A4 46                    INC    SI           ; Increment pointer.
                        ;
                        ;
                        ; Check slave CPU request stop the printer.
                        CCLRBUF:
00A5 B000                  MOV    AL,0
00A7 F086060700          LOCK XCHG AL,SPR_FLAG  ; Check flag.
00AC 3CFF                  CMP    AL,0FFH      ; If not FFH -
00AE 7503    00B3          JNZ    CONTCHK      ; jump.
00B0 E971FF  0024          JMP    CLRPRBUF     ; Clear buffer.
                        ;
                        CONTCHK:
00B3 E973FF  0029          JMP    CCOININ
                        ;
                        ;-----------------------------------------------
                        LPOUT:
                        ;
                        ;  打印输出程序
00B6 C3                    RET
                        ;
                        ;-----------------------------------------------
                        INITIO:
00B7 FA                    CLI
                        ;
                        ; Initialise seriol I/O controller.
                        ;  (TTY and console.)
00B8 2E8B160003            MOV    DX,CS:TTY_DATA   ; Initialise TTY.
00BD 2E8B1E0203            MOV    BX,CS:TTY_CMND
00C2 2E8A0E0403            MOV    CL,CS:TTMODE
00C7 2E8A2E0503            MOV    CH,CS:TTCMND
00CC E81B00  00EA          CALL   INITSIO
00CF 2E8B160603            MOV    DX,CS:CON_DATA   ; Initialise -
00D4 2E8B1E0803            MOV    BX,CS:CON_CMND   ; console.
00D9 2E8A0E0A03            MOV    CL,CS:COMODE
00DE 2E8A2E0B03            MOV    CH,CS:COCMND
00E3 E80400  00EA          CALL   INITSIO
                        ;
00E6 E81800  0101          CALL   INITINT          ; Initialise -
                        ;                          ; interrupt.
00E9 C3                    RET
                        ;
                        INITSIO:
00EA EC                    IN     AL,DX -   ; Dummy read.
00EB EC                    IN     AL,DX
00EC EC                    IN     AL,DX
00ED 87D3                  XCHG   DX,BX     ; Now pointer is -
00EF EC                    IN     AL,DX     ; command.
00F0 B041                  MOV    AL,041H   ; Dummy set.
00F2 EE                    OUT    DX,AL
00F3 B081                  MOV    AL,081H
00F5 EE                    OUT    DX,AL
00F6 EE                    OUT    DX,AL
00F7 B041                  MOV    AL,041H
```

```
00F9 EE                 OUT     DX,AL
00FA 8AC1               MOV     AL,CL       ; Set mode.
00FC EE                 OUT     DX,AL
00FD 8AC5               MOV     AL,CH       ; Set command.
00FF EE                 OUT     DX,AL
0100 C3                 RET
                ;
                INITINT:
                ; Initialise interrupt controller.
0101 B013               MOV     AL,ICW1     ┐
0103 E6C0               OUT     INTCW0,AL   │
0105 B020               MOV     AL,ICW2     │
0107 E6C2               OUT     INTCW1,AL   │
0109 B00D               MOV     AL,ICW4     │ 按此顺序初始化
010B E6C2               OUT     INTCW1,AL   │
010D B03F               MOV     AL,OCW1     │
010F E6C2               OUT     INTCW1,AL   ┘
                ;
                ; Set interrupt type vector for this program.
0111 1E                 PUSH    DS
0112 B80000             MOV     AX,0    ; Set data segment.
0115 8ED8               MOV     DS,AX
0117 BB9800             MOV     BX,INT_OFST ; Set pointer.
                ; Set interrupt procedure address.
011A C70700F0           MOV     WORD PTR [BX],SEG INTRUPT6
011E C747022F01         MOV     WORD PTR [BX]+2,OFFSET INTRUPT6
0123 C7470400F0         MOV     WORD PTR [BX]+4,SEG INTRUPT7
0128 C747063601         MOV     WORD PTR [BX]+6,OFFSET INTRUPT7
012D 1F                 POP     DS
                ;
012E C3                 RET
                ;
                ;+++++++++++++++++++++++++++++++++++++++++++++++++
                ; Interrupt procedure.
                ;+++++++++++++++++++++++++++++++++++++++++++++++++
                INTRUPT6:
012F FA                 CLI
                ;
                ;
0130 B020               MOV     AL,EOI          ; Reset IRC.
0132 E6C0               OUT     INTCW0,AL
0134 FB                 STI
0135 CF                 IRET
                ;
                INTRUPT7:
0136 FA                 CLI
                ;
                ;
0137 B020               MOV     AL,EOI          ; Reset IRC.
0139 FB                 STI
013A CF                 IRET
                ;
                ;+++++++++++++++++++++++++++++++++++++++++++++++++
                        ORG     IO_TABLE
                ;
                ; I/O address data.
```

```
0300 D900   TTY_DATA    DW    TTY_BASE   ; TTY data port.
0302 DB00   TTY_CMND    DW    TTY_BASE+2 ; TTY command port
0304 CE     TTMODE      DB    0CEH       ; TTY mode
            ;  (2-stop bit, No parity, 8-bit data, *16 baud rate)
0305 25     TTCMND      DB    025H       ; TTY command
            ;  (RTS = 1, DTR = 0, RX & TX enable)
            ;
0306 D800   CON_DATA    DW    CON_BASE   ; Console data port.
0308 DA00   CON_CMND    DW    CON_BASE+2 ; Console command prot
030A CE     COMODE      DB    0CEH       ; Console mode
            ;  (2-stop bit, No parity, 8-bit data, *16 baud rate)
030B 25     COCMND      DB    025H       ; Console command
            ;  (RTS = 1, DTR = 0, RX & TX enable)
            ;
                        ORG   RAM_TABLE
            ;
0420 0000   SEG_SLAVE   DW    00000H
            ;
            ;++++++++++++++++++++++++++++++++++++++++++++++++++
            ; Data area
            ;++++++++++++++++++++++++++++++++++++++++++++++++++
            ;
  0100                  DSEG  DATA_SEG
                        ORG   DATA_OFST
            ; Slave mode identfy flag.
0000 00     SLV_FLAG    DB    0
            ;
            ; Console input critical buffer
0001 0000   B_CO_IN     DW    0
            ; Console output critical buffer
0003 0000   B_CO_OUT    DW    0
            ; Lineprinter output critical buffer
0005 0000   B_LP_OUT    DW    0
            ; Flag to stop print.
0007 00     SPR_FLAG    DB    0
            ;
  0003      L_CRTBUF    EQU   (OFFSET $)-(OFFSET SLV_FLAG)
            ;
            ;++++++++++++++++++++++++++++++++++++++++++++++++++
            ; Extra data area and statck arae
            ;++++++++++++++++++++++++++++++++++++++++++++++++++
            ;
  0200                  ESEG  L_BUF_SEG
                        ORG   L_BUF_OFST
0000                    RB    L_LP_BUF
            ;
  0000      B_LP_BUF    EQU   L_BUF_OFST ; Bottom of buffer.
  1000      T_LP_BUF    EQU   OFFSET $   ; Top of buffer
            ;
  0200                  SSEG  L_BUF_SEG
                        ORG   T_LP_BUF
1000 0002               DW    L_STACK
            ;
            STACK_TOP:
            ;
            ;
                    END
```

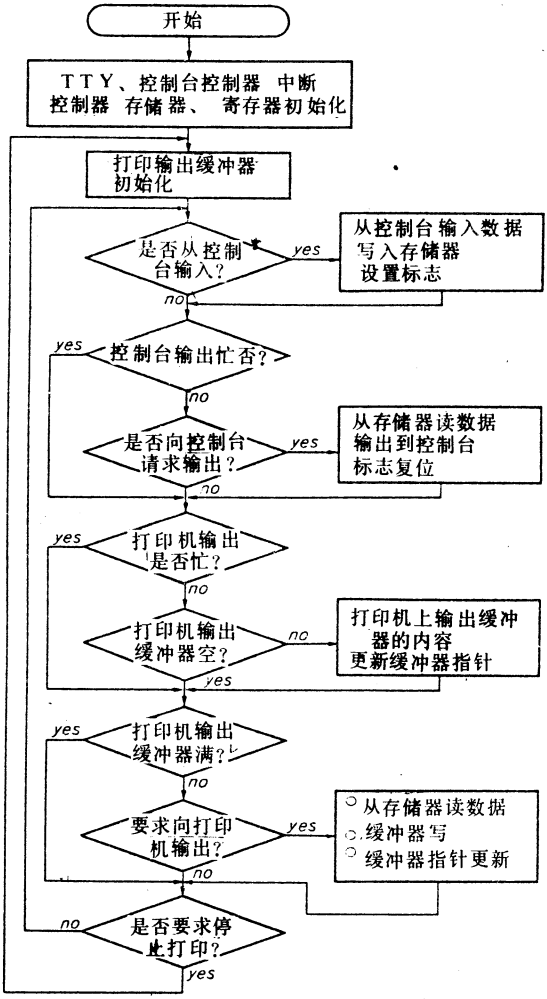


图5.1 程序例13的流程图

6 多总线

多总线是INTEL公司首先在SBC单板计算机系列上采用的总线，以后NS、AMD、三菱电机等公司也出售了多总线用的兼容插件板。

8086微处理器外围的总线控制器8288和总线仲裁器8289都是按多总线的规格设计的芯片。因此，在采用8086芯片设计单板计算机时，模板间的总线（系统总线）最好要符合多总线的规格。多总线虽然也规定了模板的尺寸大小，但是对连接器和模板大小的规定并不很严格，因此有些用户在使用时，只是信号线的规格符合多总线的要求，而连接器和模板的尺寸却采用独自的规格。

多总线已由IEEE加以标准化，也称为IEEE-796总线。

多总线具有下述特点：

- 能很容易组成多处理器系统。
- 在各总线主控制器[注]上都有总线仲裁器。
- 可以混用8位处理器和16位处理器。
- 可以有多重的8级中断请求信号。
- 信号线由负逻辑定义。

6.1 信 号 线

表6.1和表6.2列出了多总线的信号线定义。多总线的信号线在

（注） 可输出地址和命令，对总线进行控制的控制器称为总线主控制器，如CPU和DMA控制器；而从总线主控制器接受命令，将数据输出给总线的控制器称为总线从属控制器，如存储器和I/O控制器。

表6.1 P1的信号定义

内容	插脚编号	器件面 信号名	器件面 功能	插脚编号	焊接面 信号名	焊接面 功能
电源	1	GND	信号地	2	GND	信号地
	3	+5V	+5V电源	4	+5V	+5V电源
	5	+5V		6	+5V	
	7	+12V	+12V电源	8	+12V	+12V电源
	9		未定义（注2）	10		未定义（注2）
	11	GND	信号地	12	GND	信号地
总线控制	13	BCLK*	总线时钟	14	INIT*	初始化
	15	BPRN*	总线优先级输入	16	BPRO*	总线优先级输出
	17	BUSY*	总线忙	18	BREQ*	总线请求
	19	MRDC*	存储器读命令	20	MWTC*	存储器写命令
	21	IORC*	I/O 读命令	22	IOWC*	I/O写命令
	23	XACK*	传送响应	24	INH1*	禁止1（RAM禁止）
总线控制及地址	25	LOCK*	闭锁（注3）	26	INH2*	禁止2（ROM禁止）
	27	BHEN*	高位字节允许	28	AD10*	地址总线
	29	CBRQ*	公共总线请求	30	AD11*	
	31	CCLK*	固定时钟	32	AD12*	
	33	INTA*	中断响应	34	AD13*	

续表 6.1

内容	插脚编号	器件面 信号名	器件面 功能	插脚编号	焊接面 信号名	焊接面 功能
中断	35	INT6*	中断请求	36	INT7*	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*	

续表 6.1

内　容	插脚编号	器件面 信号名	功　能	插脚编号	焊接面 信号名	功　能
	69	DAT4*		70	DAT5*	
	71	DAT2*		72	DAT3*	
	73	DAT0*		74	DAT1*	
电源	75	GND	信号地	76	GND	信号地
	77		未定义	78		未定义
	79	－12V	－12V电源	80	－12V	－12V电源
	81	＋5V	＋5V电源	82	＋5V	＋5V电源
	83	＋5V		84	＋5V	
	85	GND	信号地	86	GND	信号地

（注1）* 表示负逻辑信号。

（注2）有时插脚9和插脚10定义为－5V电源（INTEL公司定义的信号，在IEEE标准中未采用）。

（注3）在IEEE方案中增加的信号。

表6.2 P2的信号定义

内容	插脚编号	器件面 信号名	器件面 功能	插脚编号	焊接面 信号名	焊接面 功能
	1	GND	信号地	2	GND	信号地
	3	+5VB	+5V电池电源	4	+5VB	+5V脉冲电源
	5			6	VCCPP	+5V电池电源
	7	-5VB	-5V电池电源	8	-5VB	-5V电池电源
	9			10		
插脚1至插	11	+12VB	+12V电池电源	12	+12VB	+12V电池电源
脚38是根据	13	PFSR*		14		
INTEL 公	15	-12VB	-12V电池电源	16	-12VB	-12V电池电源
司规格定义	17	PFSN*	掉电检测	18	ACLO	AC电源低
的，IEEE	19	PFIN*	掉电故障中断	20	MPRO*	存储器保护
还没有实行	21	GND	信号地	22	GND	信号地
标准化。	23	+15V	+15V电源	24	+15V	+15V电源
	25	-15V	-15V电源	26	-15V	-15V电源
	27	PAR1*	奇偶校验中断1	28	HALT*	总线主设备暂停
	29	PAR2*	奇偶校验中断2	30	WAIT*	总线主设备等待
	31			32	ALE	总线主设备允许地址锁存
	33			34		

续表 6.2

内　容	插脚编号	器件面 信号名	器件面 功能	插脚编号	焊接面 信号名	焊接面 功能
	35			36	BD REST*	模板复位开关
	37			38	AUX RESET*	系统复位开关
	39			40		
	41			42		
	43			44		
	45			46		
	47			48		
	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			60		

（注1）* 表示负逻辑信号。

（注2）空白表示还未定义。

P1插头（86条插脚、插脚的间距为3.96mm的印制板插头）和P2插头（60条插脚、插脚的间距为2.54mm的印制板插头）上定义。P1连接器是主插头，定义总线的主要信号。P2插头只有ADR14*～ADR17*的地址总线现在已由IEEE进行了标准化，其他的信号还有待以后定义。但是，INTEL公司和其他很多模板厂家生产的模板，都使用表6.2所示的信号。这些信号是根据INTEL公司的原始规格定义的，但IEEE还没有进行标准化。

（1）数据总线

数据总线包括DAT0*～DATF* 共16条。*表示是负逻辑信号。在8位存取时，采用DAT0*～DAT7*。在多总线上，通过字节交换功能，可以在同一块模板上进行16位和8位两种存取（参见6.2.2）。

（2）地址总线

地址总线包括ADR0*～ADR17* 共24条。由于ADR14*～ADR17*是在其他地址总线以后定义的，所以被分配在P2上。现在市場上出售的大多数模板只考虑了16位或20位的地址，但是，今后24位地址的模板将会占主要地位。除此之外，在地址总线中还有BHEN*（高位字节允许）信号，在字节交换时使用。该信号与8086芯片的BHE（高位总线允许）信号定义不同，一定不要弄错。

（3）中断的有关信号

中断的有关信号包括INT0*～INT7*的8级中断請求线和INTA*（中断允许）信号。INT0*～INT7*是集电极开路输出的，在一条线上可以加多重中断。8个信号中INT0*的优先级最高。INTA*是中断响应信号，在向量中断时使用（参见6.2.4）。

（4）命令信号

有关命令的信号包括下面6个信号：

MRDC*　存储器读命令

MWTC*　存储器写命令

IORC＊　I/O读命令
IOWC＊　I/O写命令
XACK＊　传送响应
LOCK＊　闭锁

这里的XACK＊（传送响应）是表示数据的读、写已经结束的信号，在该信号变成有效之前CPU必须等待。LOCK＊信号可作为两块RAM板的$\overline{\text{LOCK}}$信号使用（参考1.5.3和6.2.6）。

（5）总线仲裁信号

这些信号是为了裁决总线使用权的信号，包括下述几种信号。详细内容請参考6.2.3。

BCLK＊　总线时钟
BREQ＊　总线请求
BPRN＊　总线优先权输入
BPRO＊　总线优先权输出
BUSY＊　总线忙
CBRQ＊　公共总线请求

（6）P1的其他信号

除上述的信号外，还有如下信号：

INIT＊　初始化
该信号是使系统复位的信号
CCLK＊　固定时钟
这是通用的时钟信号。
INH1＊　RAM禁止
INH2＊　ROM禁止

（7）P1的电源

电源由地线、＋5V、＋12V、－12V等进行定义。以前把插脚9和插脚10定义为－5V，但现在这两个插脚未定义，是空插脚。有时在动态RAM板上还需要－5V的电源。

（8）P2的信号线

P2中除了4条地址总线外，在IEEE标准中均未定义。但INTEL公司的SBC单板上却按表6.2，对插脚进行了定义。这些信号可以分为以下几类：

● 电源

定义了±15V和备用电池电源。

● 电源断电故障信号

这此信号是断电时的支援信号，請参考6.2.5。

● 奇偶校验中断

这是存储器、总线的奇偶校验错误的中断請求信号，共定义了两条。

● 主处理器辅助信号

包括主处理器的状态信号和复位信号。AUX RESET 信号与复位开关连接。主处理器通过该信号产生INIT信号输出到P1插头。BD RESET是只复位与此相连接模板的复位信号、不产生INIT信号。

6.2 总线的结构

6.2.1 读、写信号的波形

图6.1、图6.2分别为读信号、写信号的波形。从图中可以看出，地址和写数据必须在命令有效之前至少50nS就已经稳定，并在命令无效后至少还要稳定50nS以上才行，这是作为主控制器的必要条件。

读数据和XACK*信号必须在命令结束后最长65nS以内，就要变为浮动状态，这是从属控制器的必要条件。

6.2.2 字节交换

在多总线系统中，可以允许16位处理器和8位处理器混合使用。为此，在有16位数据总线的主控制器、从属控制器板上安装了交换缓冲器。在只存取数据的高8位时，可通过交换缓冲器，将高

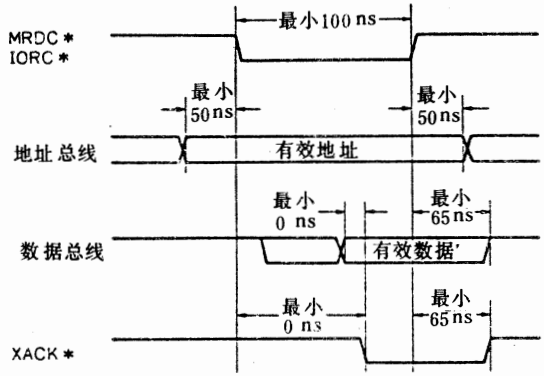


图6.1　多总线的读信号波形

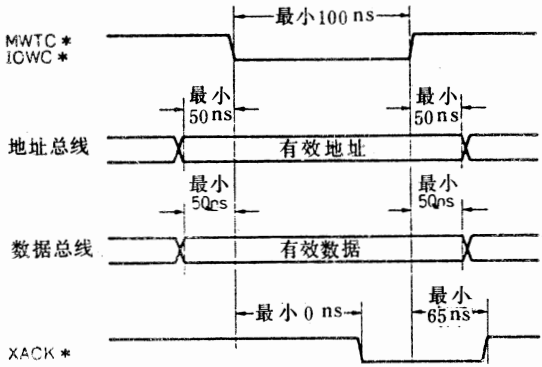


图6.2　多总线的写信号波形

8位的数据输出到DAT0*～DAT7*线上，再借助于 DAT0*～DAT7*进行数据的传输。图 6.3 表示字节交换的数据传输路径，图中的虚线就是上面讲的路径。

表6.3表示了字节交换和ADR0*、 BHEN*信号间的关系。从表中可以看出字节交换是由ADR0* 和BHEN*信号控制的。BHEN*是表示进行16位存取的信号，与8086的BHE 信号（BHE信号表示存取高位字节）的定义不同。

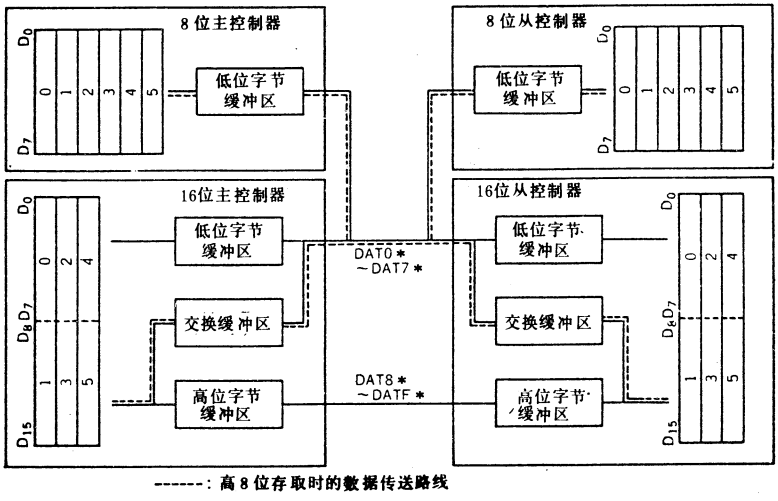


图6.3 字节交换的数据传输路径

表6.3 字节交换与ADR0*、BHEN*的关系

存取形式	8位主控制器		16位主控制器		8位从属控制器		16位从属控制器	
	ADRO	BHEN	ADRO	BHEN	ADRO	BHEN	ADRO	BHEN
存取低8位	0	0	0	0	0	忽略	0	0
存取高8位	1	0	1	0	1	忽略	1	0
存取16位	不可能		0	1	不可能		0	1

图6.4是交换缓冲器的逻辑电路图。表6.4是该逻辑电路的真值表。

6.2.3 总线仲裁

多总线系统上的仲裁，采用下述信号，以同步方式进行。

（1）BLCK*（输入）

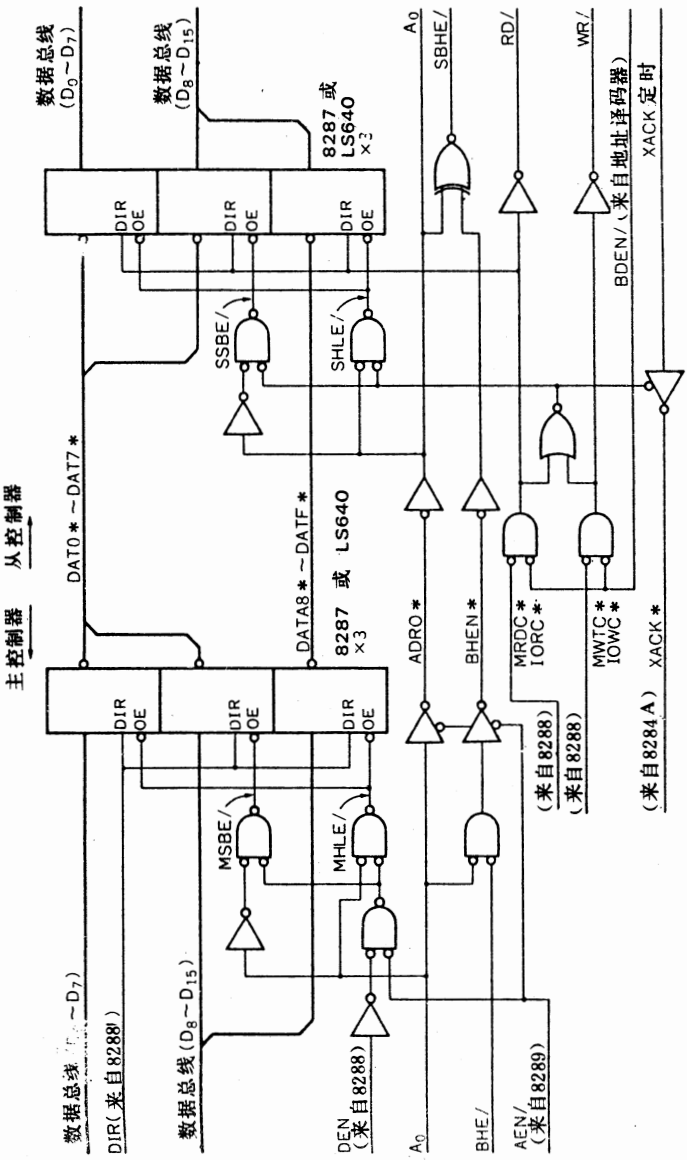


图6.4　多总线的字节交换缓冲器电路图

表6.4 图6.4电路的真值表

8086信号		主控制器信号			从属控制器信号			存取形式
A_0	BHE/	BHEN	MSBE/	MHLE/	SBHE/	SSBE/	SHLE/	
L	H	L	H	L	H	H	L	存取低8位
H	L	L	L	H	L	L	H	存取高8位
L	L	H	H	L	L	H	L	存取16位

BLCK*是同步用的时钟信号。

（2）BREQ*（输出）

BREQ*是表示总线請求的信号，它能阻止优先级低的主控制器获得总线。

（3）BPRN*（输入）

BPRN*信号表示优先级高的主控制器有总线請求。

（4）BPRO*（输出）

BPRO*是BPRN*与BREQ*两信号取“或”的信号，用它表示优先级高的主控制器对优先级低的主控制器有总线請求。该信号连接在优先级低的主控制器BPRN*上。

（5）BUSY*（输入输出）

BUSY*是某一个主控制器表示它已获得了总线的信号。该信号有效时，其他的主控制器就不能获得总线。该信号是集电极开路输出的。

（6）CBRQ*（输入输出）

CBRQ*信号表示某个主控制器有总线請求。优先级高的主控制器的請求可以用BPRN*信号检出，但是优先级低的主控制器的請求要由CBRQ*信号进行检出。CBRQ*信号为集电极开路输出。

图6.5是仲裁器的基本电路，图中还表示了总线获得和释放的条件。請求FF触发器电路是使模板内部来的总线請求，与BCLK*

同步的触发器，其输出信号BPRO*、BREQ*、CBRQ*将输出给总线。获得FF电路是表示是否获得总线的触发器。当有总线請求并且

- 总线处于关闭状态（BUSY* = H）
- 优先级高的主控制器沒有总线請求（BPRN* = L）

时，该触发器置位。在向总线输出地址、数据、命令时，该信号作为响应信号。（命令信号至少在地址、数据响应之后50nS后才能变成有效。）获得FF电路在模板内部来的总线請求变为无效之前不能复位。

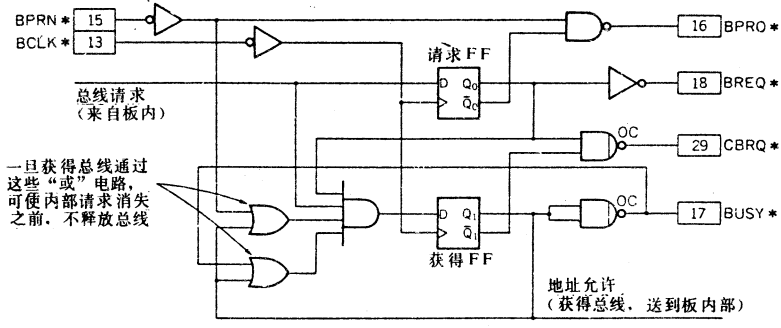


图6.5 多总线仲裁器的基本电路

在多总线仲裁器的基本电路中，一旦获得总线，只要内部来的总线請求不变为无效就不释放总线。（在基本仲裁电路中沒有考虑CBRQ*信号）。总线仲裁器8289可以设定表3.1中所列的各种总线的获得条件及释放条件，这些条件都是通过图6.5中总线的請求信号之前，加上门电路来实现的。

图6.6是总线的获得和释放的一个例子。在该例中，开始是主控制器A获得总线，在A释放总线的同时，主控制器B获得了总线，例中假设除了A、B之外，其他的主控制器沒有总线請求。（图中的信号以正逻辑表示。）

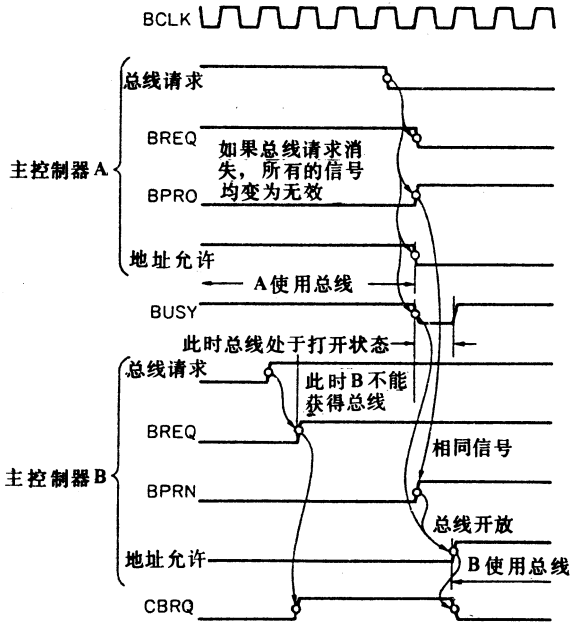


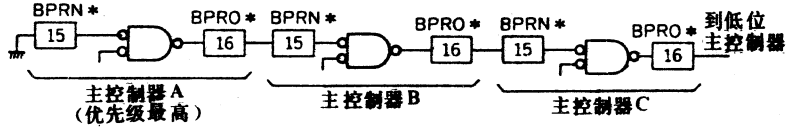
图6.6　多总线的获得、释放波形图

总线的获得、释放条件和总线的使用效率是总线仲裁器设计中最值得注意的问题。

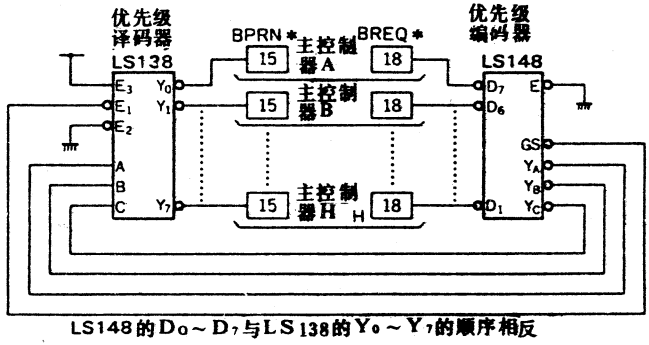
比如，多总线仲裁器的基本条件应该是“每次总线存取都要获得总线，而在存取结束时释放总线”。但是在这种方式中，总线的获得、释放频繁出现，使仲裁的时间增加，从而使总线的使用效率降低。为了提高使用效率，8289芯片中采用了通常处于获得总线状态，只在其他主控制器有总线請求时才释放总线的方式。即在8289中一般采用ANYRQST为低电平的方式。但是，这种方法的缺点是使其他主控制器为获得总线所需的时间很长，主控制器的等待状态增多，在向磁盘传输数据时很容易丢失数据而产生错误（即存储器的读、写操作跟不上磁盘旋转而产生的错误）。

究竟采用哪一种方式是系统设计上的关键问题。一般说来，主控制器数量少时采用前者的方式，而主控制器数量多时采用后者的方式。

图6.7表示BREQ*、BPRN*、BPRO*信号在母板上连接的情况。除此以外的其他信号均连接到总线上。图中（a）是将优先级高的主控制器的請求，通过级联连接，传递到优先级低的主控制器的方法，这里不使用BREQ*信号。这种方法的电路简单，但如果主控制器数量很多时，则传递所花的时间很长，造成裁决错误。图中（b）是使用优先级编码器LS148芯片，并行处理各主控制器的請求的方法，这里不使用BPRO*信号。由于在这种方法中只进行一次优先级处理，所以沒有传递延迟的问题，但是需要使用LS148、LS138两个芯片。



（a）



（b）

图6.7　母板上BREQ*、BPRN*、BPRO*信号的连接
（a）级联连接方法处理优先级（b）用优先级编码器处理优先级

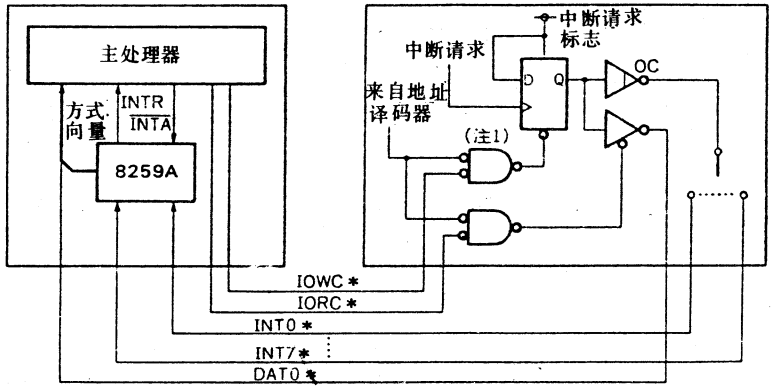
6.2.4　中断

中断請求通过INT0*～INT7*信号传给处理器。INT0*～

INT7*为集电极开路的输出信号，可以进行多重中断。接受中断的方法包括非总线向量中断和总线向量中断两种。

（1）非总线向量中断

非总线向量中断就是在发生中断时，只有INTn*(n=0～7)变为有效信号，而其他信号线上无任何输出。中断的类型向量（在8085A中是调用地址）由处理器模板内部的中断控制器产生。这是一种最简单的中断，只要不是多重中断，它只能接受8级以下的中断請求。在多重中断时应该采用图6.8所示的电路，这时处理器可以读出哪一块板上发生了中断。在处理器模板上，只要把INT0*～INT7*连接到中断控制器的中断請求输入端即可。这里不使用INTA*信号。



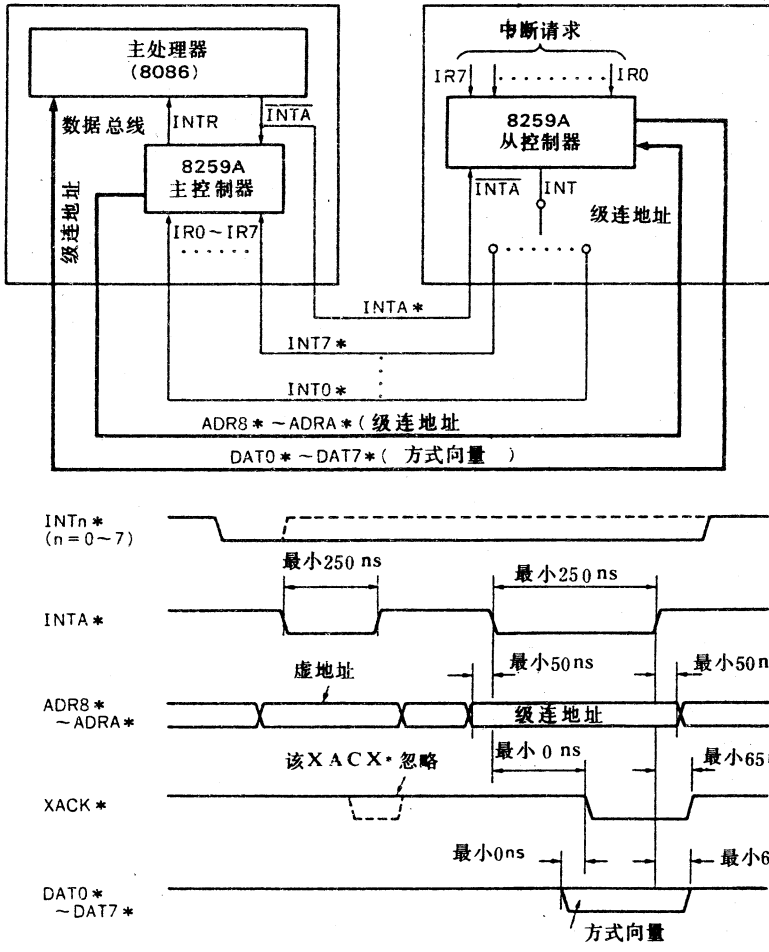
图中：（注1）中断请求标志也可以由IORC复位。

（注2）在多总线中的硬件组成要使处理器能读取中断请求标志的内容。

图6.8 非总线向量中断的电路图

（2）总线向量中断

总线向量中断是将类型向量（8085A中是调用地址）输出到DAT0*～DAT7*的中断。在处理器模板上有主中断控制器8259A，而在請求中断模板上有从属中断控制器8259A时，应使用这种中断方式(参阅3.4)。



图中：

总线向量中断时，在DAT0*～DAT7上输出类型向量。

图6.9 总线向量中断的波形图

图6.9是8086总线向量中断的信号连接图和波形图。从波形图上可以看出，首先在INTn＊（n＝0～7）上产生中断請求。在处理器板上，INTA＊两次变成有效，作为对中断的响应。第1个INTA＊信号，只起向从属控制器8259A通知“这是第1个INTA＊信号”的作用。当第2个INTA＊信号变为有效时，处理器板把板主控制器8259A的级联地址输出到ADR8＊～ADRA＊。請求中断的从属控制器8259A与该级联地址相比较，如果二者一致就将类型向量输出给DAT0＊～DAT7＊。处理器（8086）读出该向量，并分支转移到所规定的中断处理程序。由于在总线向量中断中，一条INT输入可以输出3位级联地址，所以总线向量中断一共可以处理8×8＝64级中断請求。

上面所述的例中，处理器采用8086芯片。如果处理器采用8085A时，INTA＊信号要3次变为有效，在第2、第3个INTA＊有效时，将调用地址输出给数据总线。

在总线向量中断时，如上所述，所用的处理器不同、信号波形也不同。因此，所采用的处理器种类不同，中断处理的硬件也是不同的。也就是说，为8086设计的模板不能在8085A的中断处理中使用。

6.2.5 禁止操作

在按存储器编址的I/O或ROM和RAM分配在同一地址时，使用禁止操作。例如，引导用的ROM在系统总线（多总线）上时，复位以后马上就选择ROM，而在把主程序传送到RAM区域后就必须选择RAM。这时从ROM模板上把禁止选择RAM的信号输出给INH1＊。

图6.10为禁止操作的波形图。图中ROM和RAM被分配在同一地址，通过从ROM输出的禁止信号以禁止选择RAM。首先通过处理器发出的命令，使ROM、RAM两方面的选择信号都变为有效。但是，对RAM来说，在命令发出以后的t_{XACKA}（最小50nS）期间，数据总线、XACK＊信号不能变为有效。而对ROM来说，为

了使INH1＊信号在 t_{XACKA} 期间内变为有效，必须在有效地址信号开始后100nS内INH1＊信号就得变为有效。INH1＊信号有效后，RAM的选择即被禁止。INH1＊信号必须有足够长的有效期，以免禁止的RAM再次被选择。ROM的XACK＊信号至少在命令发出后延迟1.5μS才能变为有效。

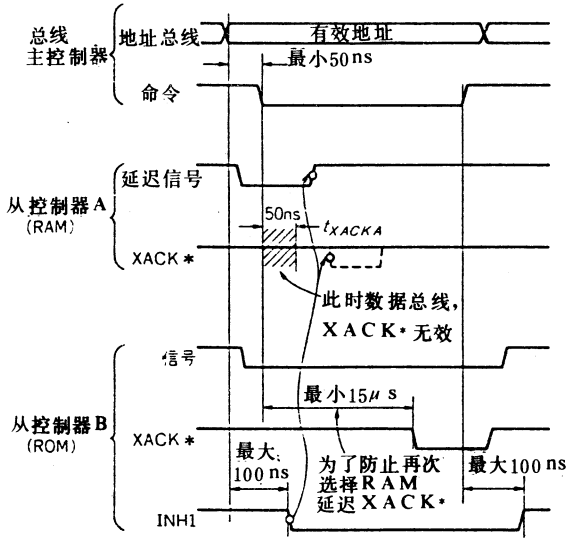


图6.10　禁止操作的波形图

上面叙述了禁止操作的概要。但是，在采用8086作为处理器时，由于地址空间扩大了，而且处理器模板内部有引导ROM，所以很少使用禁止操作。

6.2.6　电源断电时的时序信号

在INTEL公司的规格中，为了处理断电的情况，在P2上定义了以下的信号。图6.11是这些信号的波形图。

（1）ACLO

ACLO是指示交流电源被切断的信号，是下述各信号的驱动信号。ACLO信号为正逻辑。

（2）PFSN＊

PFSN＊是断电的检测信号，锁存之后输出。PFSN＊信号的复位由PFSR＊进行。

（3）PFIN＊

PFIN＊是断电中断請求信号。

（4）MPRO＊

MPRO＊是断电时的存储器保护信号。该信号用于电池作后备电源时的存储器保护。

（5）PFSR＊

PFSR＊是PFSN＊锁存的复位信号。

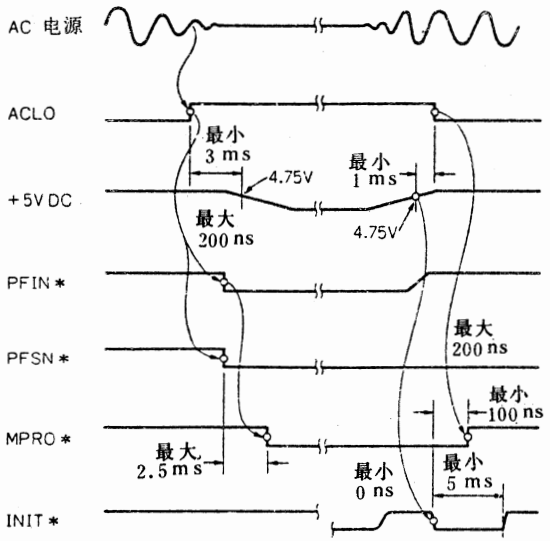


图6.11　电源断时信号的波形

6.2.7　双端口的RAM电路

这是一种在处理器模板上的RAM电路，从处理器和多总线都能存取的RAM电路称为双端口（dual port）RAM电路。图6.12就是双端口RAM电路。在多处理器系统中，处理器与处理器之间

的数据传输是通过系统存储器进行的。这时，如果传输的数据量很大，两个处理器存取系统总线的次数就要增多，总线存取的等待时间就增加，使系统的吞吐量减少。双端口RAM电路就解决了这一问题，在这种电路中，从系统总线上也可以存取处理器板上的RAM。采用双端口RAM电路在处理器之间进行数据传输时，如图6.12所示，另一个处理器就没必要存取系统总线了，从而提高了系统的吞吐量。

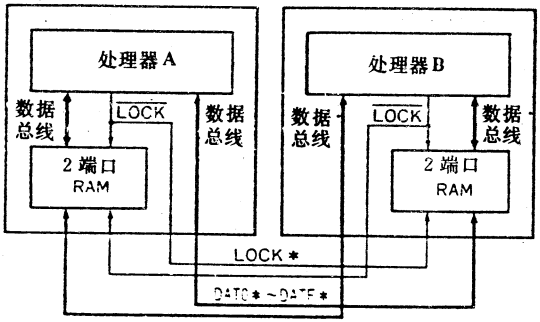


图6.12 双端口RAM电路

LOCK*信号是闭锁双端口RAM电路的信号（参阅1.5.3）。图6.13是LOCK*信号的波形图。为了使模板上的处理器在命令无效期间不存取RAM，应在命令信号变为无效之前100nS，LOCK*就要变为有效，并在下一个命令变为有效之后100nS以上都要保持有效状态。

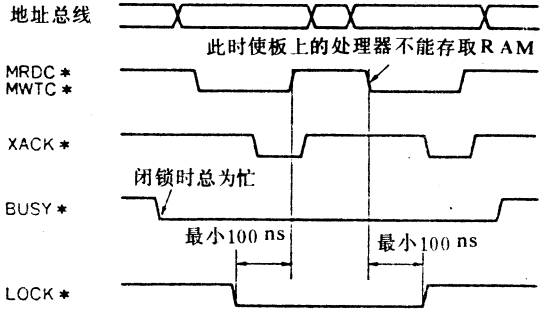


图6.13 LOCK*信号的波形

表6.5 多 总 线 的

信　号	驱　动　器（注1）（注3）				
	位　置（注4）	型　式	I_{OL} mA(最小)	I_{OH} μA(最小)	Co pF(最大)
DAT0*～DATF*	M或S	3	16	-2000	300
ADR0*～ADR13*	M	3	16	-2000	300
BHEN*	M	3	16	-2000	300
MRDC* MWTC*	M	3	32	-2000	300
IORC* IOWC*	M	3	32	-2000	300
XACK*	S	3	32	-2000	300
INH1* INH2*	禁止从属控制器的模块	OC	16	—	300
BCLK*	M(只一个)	T	48	-3000	300
CCLK*	1个	T	48	-3000	300
BREQ*	各自的M	T	5	-400	60
BPRO*	各自的M	T	5	-400	60
BPRN*	（注9）	T	5	-400	300
BUSY* CBRQ*	所有的M	OC	32	—	300
INIT*	M	OC	32	—	300
INTA*	M	3	32	-2000	300
LNT0*～INT7*	S	OC	16	—	300

（注1）驱动器 $\begin{cases} I_{OL}: \text{“L”输出电流} \\ I_{OH}: \text{“H”输出电流} \\ Co: \text{驱动电容能力} \end{cases}$　（注2）接收器 $\begin{cases} I_{IL}: \text{“L”输入电流} \\ I_{IH}: \text{“H”输入电流} \\ C_I: \text{输入电容} \end{cases}$

（注7）中心优级模块　（注8）串行优先级方式的下一个主控制器模块

直 流 特 性

接收器（注2）				终端负载		
位 置	I_{IL} mA(最大)	I_{IH} μA(最大)	C_I pF(最大)	位置	形式（注5）	电阻值（kΩ）
M或S	-0.8	125	18	1个	PU	2.2
S	-0.8	125	18	1个	PU	2.2
S	-0.8	125	18	1个	PU	2.2
S（存储器或存储器编址的I/O）	-2	125	18	1个	PU	1
S(I/O)	-2	125	18	1个	PU	1
M	-2	125	18	1个	PU	150Ω
禁止的模块	-2	50	18	1个	PU	1
M	-2	125	18	MB（注6）	PU和PD	220Ω，330Ω
不限定	-2	125	18	MB（注6）	PU和PD	220Ω，330Ω
（注7）	-2	50	18	（注7）	PU	1
（注8）	-1.6	50	18			
M	-2	50	18			
所有的M	-2	50	18	1个	PU	1
所有的模块	-2	50	18	1个	PU	2.2
S(中断请求I/O)	-2	125	18	1个	PU	1
M	-1.6	40	18	1个	PU	1

（注3）输出形式：3：三状态；T：推拉输出电路；OC：集电极开路

（注4）M/S：主控制器/从属控制器

（注5）PU/PD：上推/下拉电阻

（注6）MB：母板

（注9）并行方式中为中心优先级模块；串行方式中为前面的主控制器模块（优先级高一级的主控制器）

6.2.3 外形及直流特性

图6.14是多总线模板的外形图。表6.5列出了多总线的直流特性。

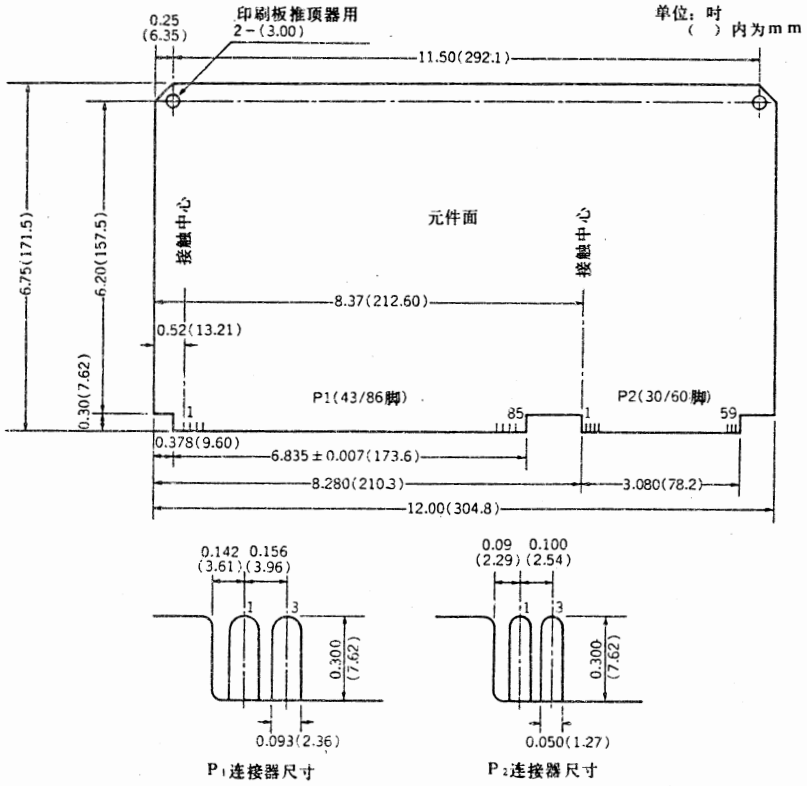


图6.14 多总线模板的外形图

6.3 从属控制器板的设计举例

图6.15是一个从属控制器模板的电路实例。图中是I/O控制器模板的总线接口部分。

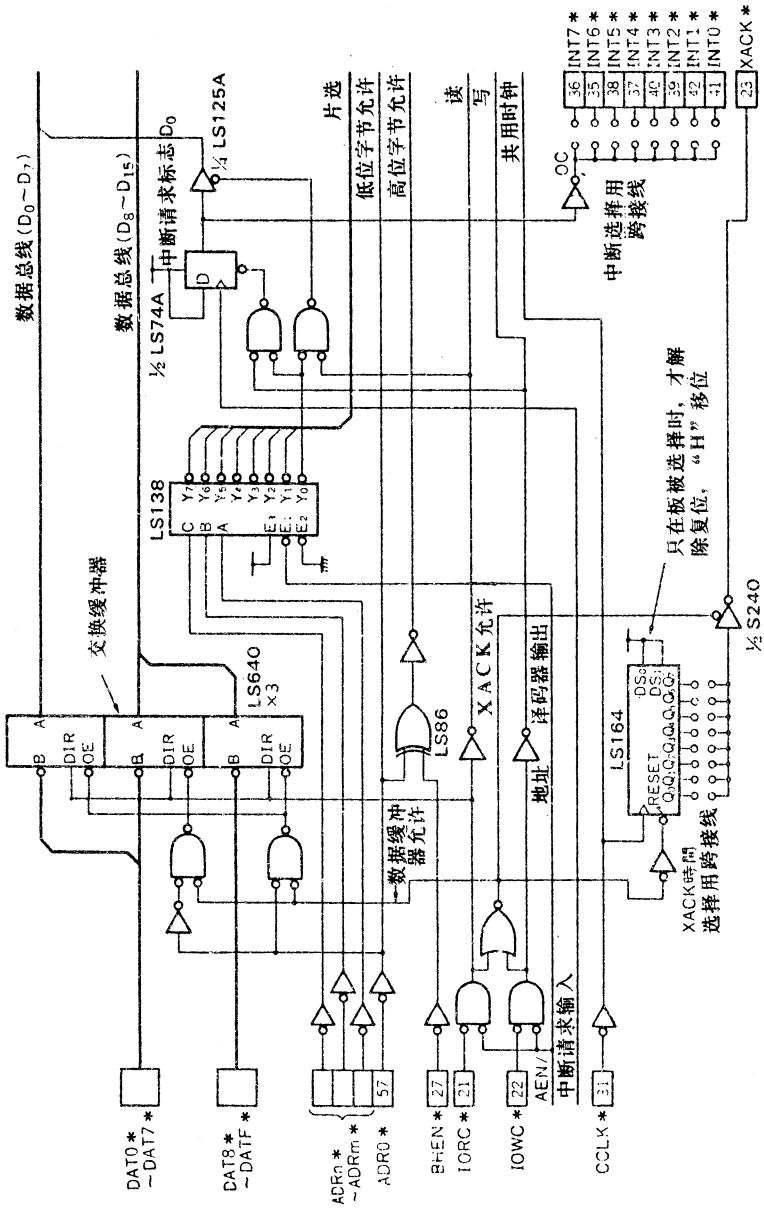


图6.15　从属控制器板的设计举例

该例的数据缓冲器与图6.4电路相同，通过ADR0信号对交换缓冲器和一般的缓冲器进行转换。IORC*、IOWC*信号与地址译码器的输出信号相“与”，只有在该模板被选择时，读、写命令才变为有效。数据缓冲允许信号和XACK允许信号是由两个命令信号取“或”后产生的。XACK允许信号通过移位寄存器LS164延迟之后产生XACK信号。LS164的时钟利用CCLK信号，采用跨接线选择定时。

芯片选择信号是通过LS138芯片将地址总线所需要的位译码后产生。低位字节允许信号是ADRO，高位字节允许信号是ADRO和BHEN的“异或”信号。中断請求标志和图6.8所示的电路相同，这里采用LS74A作为边缘检测电路。标志的內容可以通过读命令读出，标志的复位可以通过写命令进行。

7 8086的开发系统

本章将介绍8086系统的开发系统及软件支援体系，介绍内部电路仿真器（ICE86）及测试板（SDK86）。还将介绍一种8086单板机SBC86/12A。

7.1 INTEL MDS微计算机开发系统

8086/8088系统的开发系统有INTEL公司的MDS系列的开发系统，以前的MDS800和MDS系列Ⅱ是以8080/8085为基础的开发系统，但也提供了交叉汇编ASM86及交叉编译PL/M—86，还有连接程序LINK86及定位程序LOC86等。以8086为基础的MDS系列Ⅲ配有7.3Mb的硬磁盘，与系列Ⅱ相比处理速度可以提高3～5倍。

在开发8086的过程中，由于系统程序编码及开发的程序很长，因此要求使用MDS系列Ⅱ230型（软盘容量2.25Mb）以上的系统。MDS的操作系统是ISIS—Ⅱ(Intel Software Implemented Supervisor)，它不仅可对软盘文件进行格式化，还具有文件复制、删除等管理功能，以及用DEBUG命令在简单的监控程序下设定断点，进行程序的调试。如果进行更详细的分析，必须采用下述的联机仿真器（ICE86）。在ISIS—Ⅱ管理下可以使用的高级语言，除了PL/M—86以外，还提供有FORTRAN86及PASCAL—86，可以根据用户的需要进行选择。

源程序编辑用的程序包括行编辑程序（EDIT）和屏幕编辑程序CREDIT。后者的特点是可以把光标移到要修改的地方，进行简

单地修改。

由汇编语言或编译语言产生的目标程序是浮动的，可以在程序装入时决定实际的单元，在最后阶段通过其他的子程序、库程序和LINK86程序产生一个程序，然后再通过定位程序LOC86决定实际的存储单元，形成可以执行的绝对目的代码。在把它写入PROM时，要通过OH86变换成可读出的HEX文件。此外，开发系统中还有

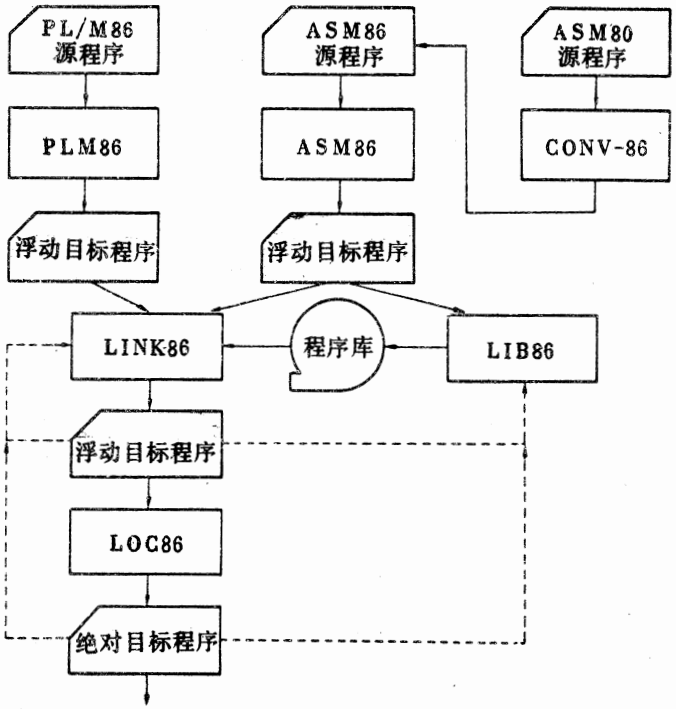


图7.1 程序开发的流程图

将8080/8085程序变换为8086用的CONV—86程序。

7.2 联机仿真器（ICE86）

在联机仿真器产生之前，开发微计算机时，先对硬件和软件分别进行开发，开发完成之后再合在一起，相互进行调试。采用联机仿真器后，可以在用户系统的硬件/软件都不完整的阶段，从MDS开发系统借用用户系统不足的存储器和I/O等来执行程序。

联机仿真器具有下述多种很强的调试功能。

（1）ICE86电缆顶端的40脚插座可取代用户系统的8086CPU，从插脚看，整个开发系统被看作有大量存储器和外围I/O装置的CPU。

（2）有存储器变换功能。用户系统工作时，可以自由选择用户系统上的存储器或开发系统内的存储器，存储器变换以1K字节为单位预先通过命令进行设定。

（3）仿真的命令包括GO、STEP。GO命令从指定的地址开始，执行到设定的断点（可设定两个断点）为止。STEP为单步执行。

（4）可以用标号指定命令中的地址，可以进行全符号调试。

（5）可以设定两个断点，通过设定跟踪信息收集的起始/结束条件，把最多约300个总线周期的信息存入跟踪存储器，以便在以后对这些内容进行检测。

此外，在联机仿真器中还有宏命令功能，宏命令就是将经常使用的命令过程预先加以定义，使用时只要给出宏命令名和最多10个参数就可以简单地指定。复合命令指的是在执行条件命令（IF）时，如果满足某一条件或者执行到指定的次数时就执行的命令。在检查跟踪存储器及存储器内的目的码时，可以进行反汇编。用汇编的助记符进行检查。

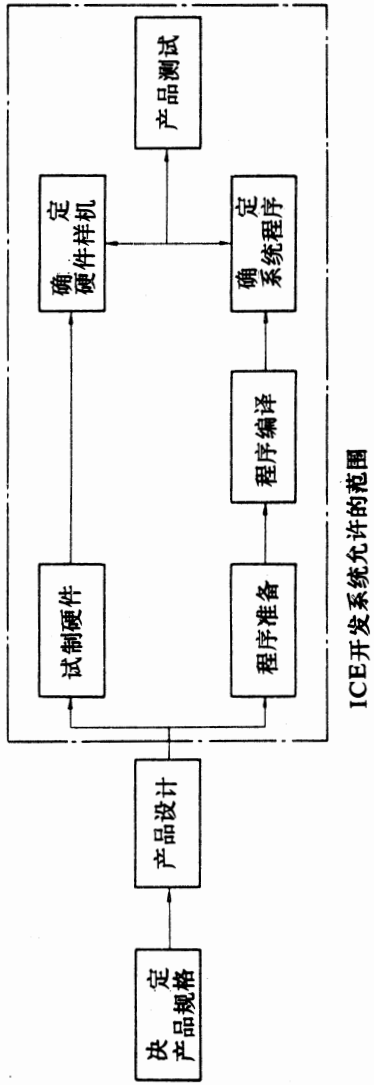


图7.2　用带ICE的开发系统开发产品的周期

7.3 检查工具（SDK86）和单板计算机（SBC86/12A）

（1）SDK86

现在已发表了8086用的系统设计工具SDK86，并附有装配手册、有系统说明和电路图的用户指南。SDK86的监控程序包括HEX键盘和8位LED显示的监控程序及连接CRT或TTY的串行监控程序，二者分别占2K字的容量。该检查工具可以执行存储器的显示/变更、模块传送、设定断点等程序，也可以执行单步程序。在串行监控程序时，通过与TTY的连接，可以用纸带阅读机和穿孔机对HEX文件进行输入/输出。其主要规格如下：

- CPU：8086 5MHz或2.5MHz
- 存储器：ROM 8K字节（2716/2316×4）
  RAM 2K字节（2142×4）
  可扩充到4K字节
- 寻址范围：ROM FE000H～FFFFFH
  RAM 0～7FF（增加RAM时可到0～FFF）
- 输入/输出：并行输入输出 48条引脚
  （8255A×2）
  串行输入输出 RS232C或电流环路（8251A）
- 中断：256级中断，可屏蔽/不可屏蔽/陷阱。

（2）SBC86/12A

SBC86/12A是使用8086的最大方式，采用总线控制器（8288）及总线仲裁器（8289）的完全兼容的单板机。板上装有32K字节的双端口RAM，不仅板上的CPU可以对存储器存取，其他的CPU也可以通过多总线对同一存储器进行存取。其规格如下：

- CPU：8086 5MHz 1.2μS指令周期
  （从队列存取时为400nS）
- 存储器：ROM 16K字节（可扩充到32K字节）

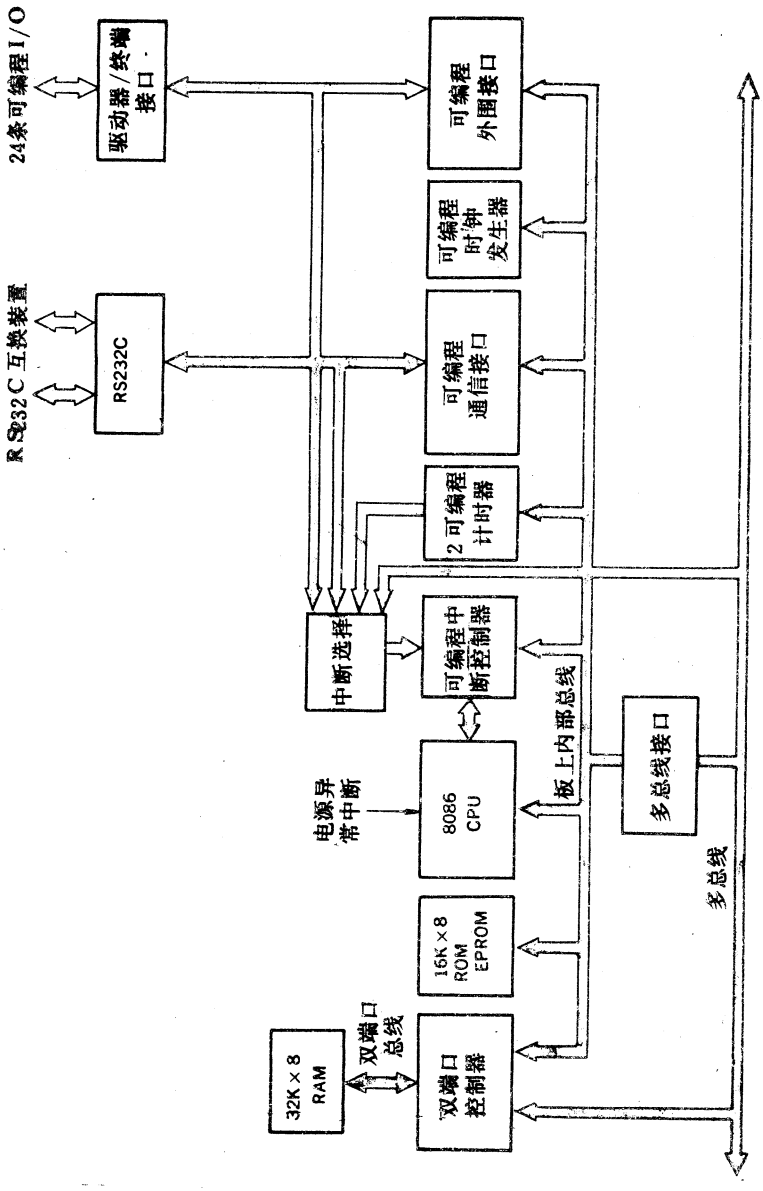


图7.3 SBC86/12A的方框图

RAM 32K字节（可扩充到64K字节）

· I/O：串行接口　RS232C（8251A）

并行接口　24条输入输出端口（8255A）

计时器　两个16位计时器（其中1个8253只作晶体振荡器使用）

· 中断　9级（装有8259A）

8 程序设计语言/实时监控程序

本章将介绍8086的开发系统MDS上可以使用的程序设计语言中的汇编语言（ASM86）和编译语言（PLM86），还将概要介绍8086用的实时监控程序（RMX86）。

8.1 汇编语言（ASM86）

8086/8088的汇编语言采用了只有高级语言中才能使用的数据结构功能，是一种接近高级语言的汇编语言。例如，在一条MOV指令中有多个变量时，程序中只指定MOV和源操作数及目标操作数即可，以后就由汇编语言根据操作数的属性产生适当的机器语言。

（1）语句

语句的格式采取

标号：（前置指令）、助记符、（操作数）；（注释）

的形式，标号在31个字符以内，为了阅读方便可以任意加空白和连接线。括号表示可选用的项目。前置指令包括段修改前缀、总线闭锁及重复前缀等，用以规定该指令后面的操作。

（2）数据的定义

数据的定义通过DB（字节）、DW（字）、DD（双字）三个汇编命令进行，用于规定开辟的区域数及其初始值等。

（3）记录

ASM86可以在符号中定义字节和字内的各位及字符串。如程序例14所示，可以将1字节分为YRS、SEX及STATUS三个段进行定义。在TEST点可以只对该部分屏蔽、取出及检查。

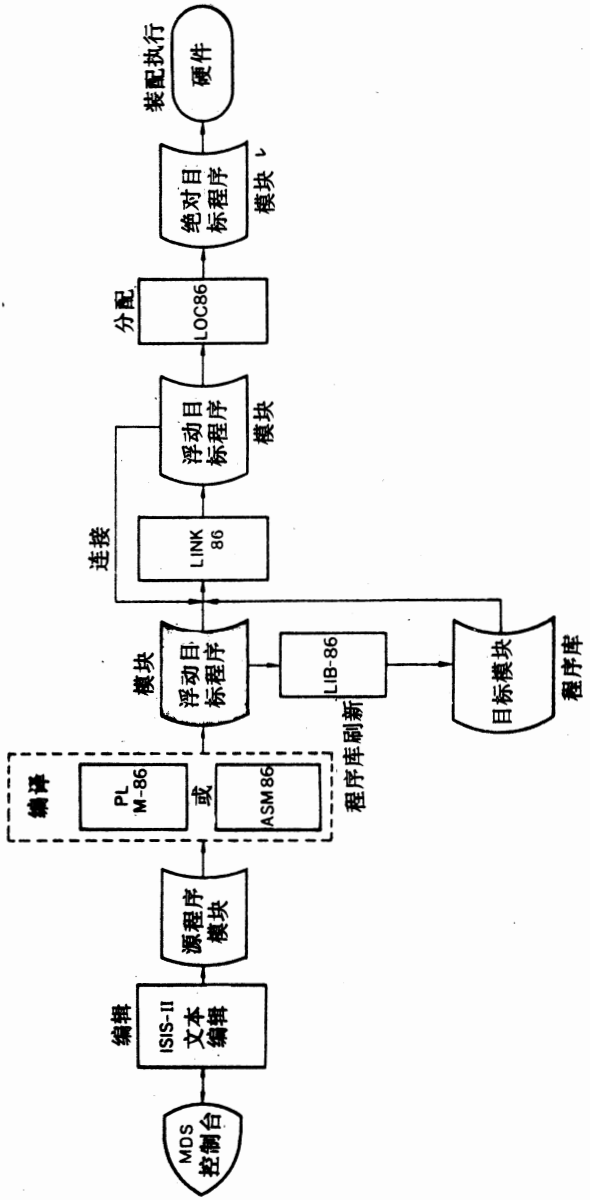

装配执行
硬件
绝对目标程序模块
分配
LOC86
浮动目标程序模块
连接
LINK 86
浮动目标程序模块
LIB-86
程序库刷新
目标模块
程序库
编译
PL M-86
或
ASM86
源程序模块
编辑
ISIS-II 文本编辑
MDS 控制台

图8.1　软件的开发过程

程序例14　ASM86记录举例

```
        EMP_BYTE DB ?                   ;1BYTE, UNINITIALIZED
        ;BIT DEFINITIONS:
        ;     7-2   :YEARS EMPLOYED
                1   :SEX(1=FEMALE)
                0   :STATUS(1=EXEMPT)
        EMP_BITSRECORD                  ;RECORD DEFINED HERE
        &          YRS_EMP:6,
        &           SEX:1,
        &           STATUS:1
        :
        ;SELECT NONEXEMPT FEMALES EMPLOYED 10+YEARS

        MOV         AL, EMP_BYTE        ;KEEP ORIGINAL INTACT
        TEST        AL, MASK SEX        ;FEMALE?
        JZ          REJECT              ;NO, QUITE
        TEST        AL, MASK STATUS     ;NONEXEMPT?
        JNZ         REJECT              ;NO, QUIT
        SHR         AL, CL              ;ISOLATE YEARS
        CMP         AL, 11              ;>=10YEARS?
        JL          REJECT              ;NO, QUIT
        ;PROCESS SELECTED EMPLOYEE
        .
REJECT: ;PROCESS REJECTED EMPLOYEE
        .
        .                               ;RECORD USED HERE
        MOV         CL, YRS_EMP         ;GET SHIFT COUNT
```

（4）结构

结构是给一组数据字段以名称和属性（长度、类型等）的程序段，由DB、DW及DD进行定义。在结构中不分配地址空间，在指令中某字段名与基地址一起访问时，结构的空间就是存储器的特定空间。如程序例15所示，在指令中访问MASTER及TXN时，结构中的RATE也一起访问。

（5）过程

在程序中反复使用同一段程序时，就把这段程序定义为过程，过程可以从程序的其他部分通过CALL调用。

8.2 PL/M-86

PL/M-86是8086/8088用的高级语言，它可以与8080/8085用的PL/M-80在源程序级向上兼容。通过使用高级语言可以缩短程

程序例15　ASM86结构的举例

```
EMPLOYEE      STRUC
  SSN         DB 9   DUP(?)
  RATE        DB 1   DUP(?)
  DEPT        DW 1   DUP(?)
  YR HIRED    DB 1   DUP(?)
  EMPLOYEE    ENDS

MASTER        DB 12 DUP(?)
TXN           DB 12 DUP(?)

;CHANGE RATE IN MASTER TO VALUE IN TXN
              MOV    AL, TXN.RATE
              MOV    MASTER.RATE, AL

 ASSUME BX POINTS TO AN AREA CONTAINING
    DATA IN THE SAME FORMAT AS THE EMPLOYEE
    STRUCTURE. ZERO THE SECOND DIGIT
    OF SSN
              MOV    SI, 1 ;INDEX VALUE OF 2ND DIGIT
              MOV    [BX].SSN[SI], 0
```

序的开发时间，减少软件的维护成本。PL/M-86可以用接近简单的英文和算术表达式进行描述。

（1）程序的组成

PL/M-86的程序能对指示数据的定义、执行的语句及注释连续进行描述。各语句用分号（；）表示结束，注释在“/*”和“*/”之间描述，可以放在程序中的任何地方。

（2）数据定义

程序通常以数据的定义开始，每个元素称为标量，名称由31个以下字符组成，可以使用五种类型，即字节、字、整型、实型及指针。表8.1是数据类型一览表。变量由说明语句定义

DECLARE　标量名　类型

数组的定义按下面形式进行，比如第6个元素可写成DATA(5)

DECLARE DATA (12)　REAL

定义相关的一组元素时采用STRUCTURE，各元素之间要加

点，如DATA·LENGTH。

```
DECLARE DATA STRUCTURE
  (LENGTH WORD, WIDTH BYTE, HEIGT WORD);
```

（3）赋值语句和各种算符

赋值语句的格式如下所示，对表达式求值后，将结果代入左边的变量中，表达式除了采用常数之外，还可以是算术运算、关系式运算和逻辑运算，表8.2为PL/M-86的表达式。

变量名=表达式

表8.1 PL/M-86的数据类型

类　型	字节数	数据范围	使　用
BYTE	1	0～255	无符号整数，字符
WORD	2	0～65535	无符号整数
INTEGER	2	-32768～$+32767$	带符号整数
REAL	4	1×10^{-38}～$3.37\times10^{+38}$	浮点数
POINTER	2/4	N/A	地址操作

表8.2 PL/M-86的表达式

表达式	算　符	结　果
算术算符	+、-、*、/、MOD	数　值
关系算符	>、<、=、>=、<=	“真”—FFH “伪”—0H
逻辑算符	AND、OR、XOR、NOT	8/16位字符串

（4）有关程序流的语句

对程序流进行控制的语句有IF语句和DO语句。在IF语句中，当满足描述的关系表达式时执行语句1，如果不满足关系表达式就执行语句2，即执行条件转移。

IF关系表达式THEN 语句1；ELSE 语句2；

DO语句就是在满足某一条件期间，反复执行某一段程序。DO语句有DO、DO CASE、DO WHILE三种形式，DO CASE表示在DO语句所描述的执行语句中，只执行满足CASE条件的部分。DO WHILE是在WHILE之后的表达式为“真”时，反复执行该段程序。

程序控制的转移语句有GOTO和CALL语句。

GOTO START

CALL 过程名（参数表）

过程在程序的开始时定义，参数表是转移到过程的变量。

（5）过程

一个复杂的程序分成若干个子程序，每个子程序就是一个过程，它具有一个功能，在主程序中可通过CALL语句调用。其使

程序例16 PL/M-86过程的程序例

```
/*DECLARATION OF A TYPED PROCEDURE THAT
  ACCEPTS TWO REAL PARAMETERS AND RETURNS A REAL VALUE*
  AVG:PROCEDURE(X, Y)REAL;
     DECLARE(X, Y)REAL,
     RETURN(X+Y)/2.0;
     END AVG;

/*ACTIVATING A TYPED PROCEDURE*/
LOW=2.0;
HIGH=3.0;
TOTAL=TOTAL+AVG(LOW, HIGH);/*2.5IS ADDED TO TOTAL*/

/*DECLARATION OF AN UNTYPED PROCEDURE
  THAT ACCEPTS ONE PARAMETER*/
TEST:PROCEDURE(X);
  DECLARE X BYTE;
  IF X=0H THEN
     COUNT=COUNT+1;
  END TEST;

/*ACTIVATING AN UNTYPED PROCEDURE*/
CALL TEST(ALPHA);/*COUNT IS INCREMENTED
                   IF ALPHA=0*/
```

用方法可参考程序例16。

8.3 实时监控程序（RMX86）

实时监控程序就是在过程控制的时间内，监视随机发生的事件，检测、存储同时产生的多个处理事项，并按优先级进行处理的程序。比如，计算机系统在处理优先级低的事件过程中，如果产生优先级高的事件，就要停止正在处理的事件，而去处理优先级高的事件，在该事件处理结束之后再返回到中断之前的事件，继续处理。RMX86不仅具有实时处理的功能，还兼有通常的通用操作系统（OS）的文件管理功能等。

RMX86的组成如图8.2所示，其核心部分包括多任务处理、多道程序处理、任务内部通信、中断处理及错误检查等基本功能，由于这些功能常驻在系统中，所以应该是只有2K字节或尽可能短的程序。基本I/O系统（BIOS）具有与I/O装置的种类和文件格式无关的数据操作功能，扩充I/O系统具有同步I/O调用和高度的作

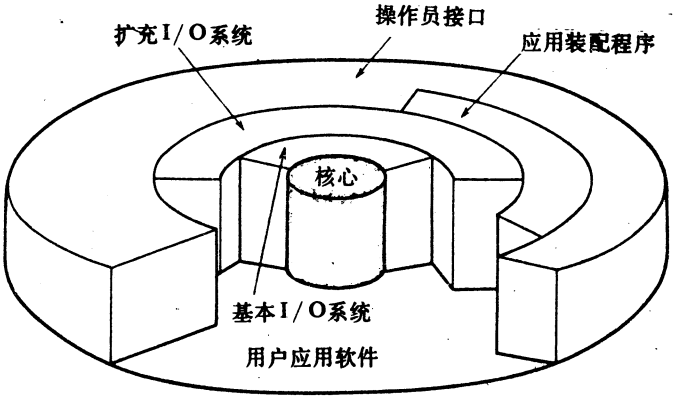


图8.2 实时监控程序RMX86的组成

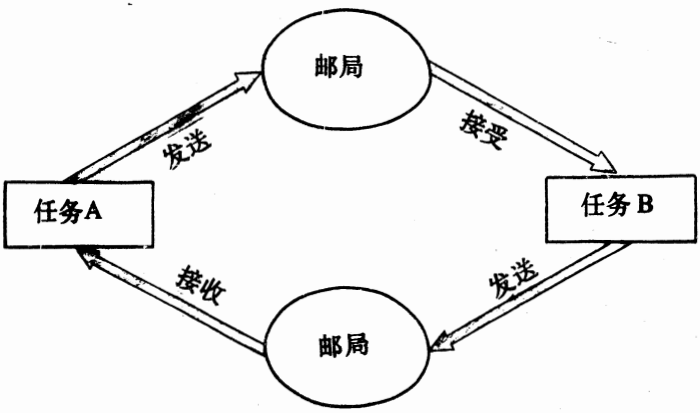


图8.3 任务A、任务B之间的信息交换

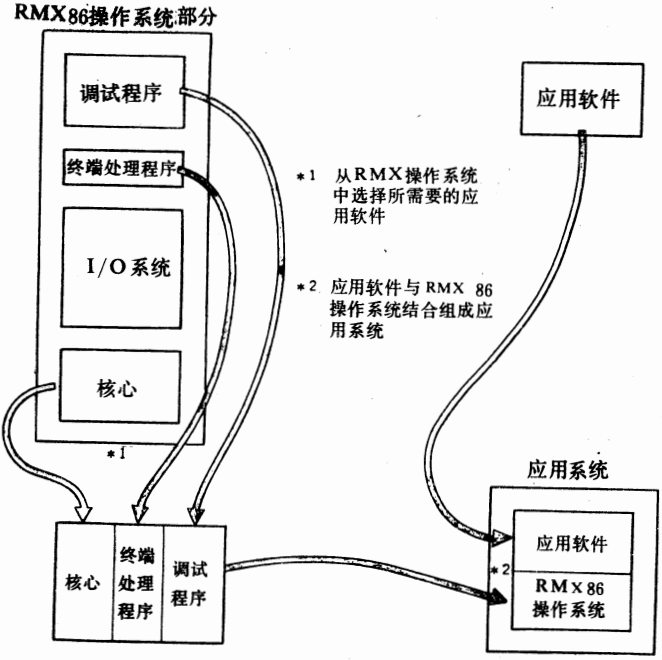


图8.4 实时监控程序（RMX86）的组成

业管理等功能。根据需要还可以选用从软磁盘、硬磁盘等大容量存储装置，将程序代码和数据装入RAM的应用装配程序以及CRT终端等的接口程序。如果需要也可把用户编写的应用程序与这些程序连接。

图8.4是RMX86操作系统中，组成应用系统的整个程序。图中除了核心部分之外都是可选程序，在不需要时就没必要连接。例如，调试程序只在系统的初始阶段及增加新功能时，做为调试使用，而在开发结束之后就可以去掉，这样可以减小应用程序的长度。任务之间的命令和数据的交换如图8.3所示，可以通过邮递的形式发出数据和接受数据。

9 系列处理器功能的扩充和8086的发展方向

以8086为中心与8087（高速运算处理器）及8089（I/O处理器）相结合，可以组成功能更强的处理器系统。这种系统可以大大提高8086的运算及I/O处理功能。本章以8086为基础对16位CPU将来的发展方向和32位CPU进行简单地介绍。

9.1 高速运算处理器（辅助处理器）8087

8087数据处理器是一种弥补8086/8088数值运算能力的处理器，它与8086组成一个系统进行工作，其连接如图9.1所示。表9.1表示

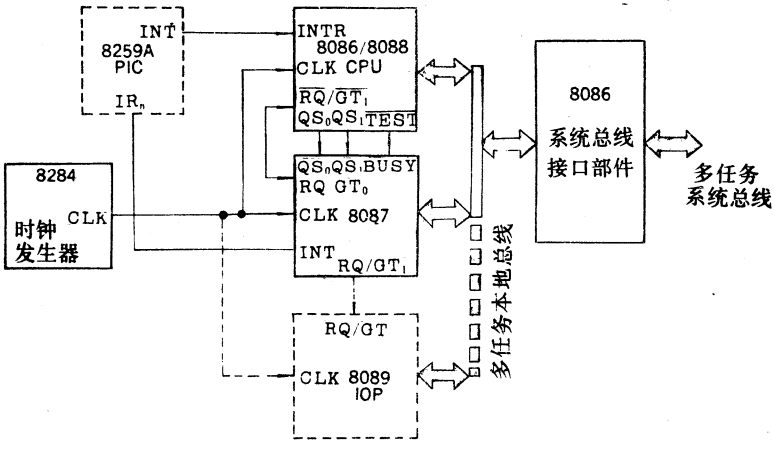
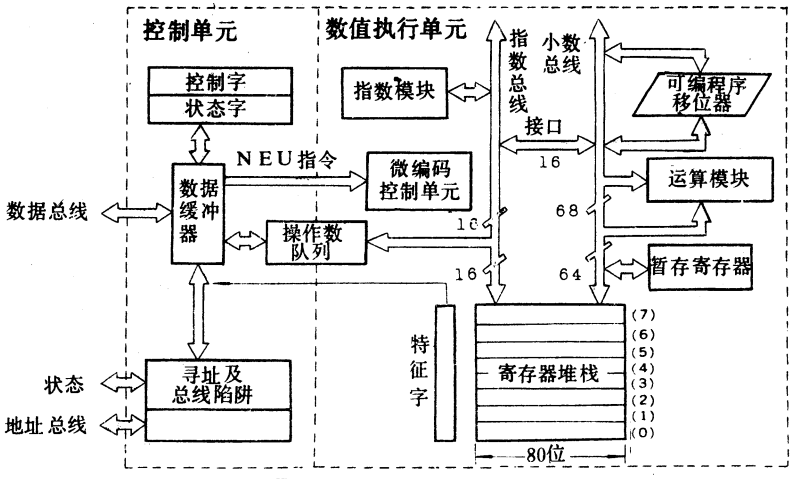


图9.1 8087的使用


控制单元
数值执行单元
控制字
状态字
指数模块
指数总线
小数总线
可编程序移位器
接口
16
NEU指令
微编码控制单元
运算模块
数据总线
数据缓冲器
操作数队列
16
68
16
64
暂存寄存器
特征字
寄存器堆栈
(7)
(6)
(5)
(4)
(3)
(2)
(1)
(0)
80位
状态
地址总线
寻址及总线陷阱

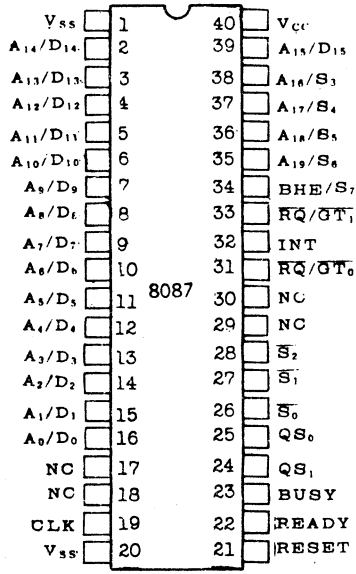

8087
1 V_{SS}
2 A_{14}/D_{14}
3 A_{13}/D_{13}
4 A_{12}/D_{12}
5 A_{11}/D_{11}
6 A_{10}/D_{10}
7 A_9/D_9
8 A_8/D_8
9 A_7/D_7
10 A_6/D_6
11 A_5/D_5
12 A_4/D_4
13 A_3/D_3
14 A_2/D_2
15 A_1/D_1
16 A_0/D_0
17 NC
18 NC
19 CLK
20 V_{SS}
40 V_{CC}
39 A_{15}/D_{15}
38 A_{16}/S_3
37 A_{17}/S_4
36 A_{18}/S_5
35 A_{19}/S_6
34 BHE/S_7
33 $\overline{RQ}/\overline{GT}_1$
32 INT
31 $\overline{RQ}/\overline{GT}_0$
30 NC
29 NC
28 $\overline{S}_2$
27 $\overline{S}_1$
26 $\overline{S}_0$
25 QS_0
24 QS_1
23 BUSY
22 READY
21 RESET

图9.2 8087的组成

增加了8087后的处理能力 及处理速度与8086软件仿真时的比较。图9.2为8087的结构框图。

8087接受CPU队列状态信息，与CPU取得同步后，只选择与自己有关的指令，进行译码。也就是说，所有8087的机器语言指令的最初5位是ESC类型的(D8H～DFH)指令，它的控制单元对其他指令均不起反应。CPU在取包括换码在内的指令时，与此并列的辅助处理器也同时取同一代码，借助于CPU的帮助，对该指令译码并执行。此后，CPU可以并行执行其他操作，当需要运算结果时，通过插入WAIT指令，与$\overline{\text{TEST}}$信号结合，在和CPU同步后就可以对结果进行传输。如果8087检测出有错误等异常状态，可以通过程序中断控制器8259A，对CPU产生中断，进行中断处理。

8087为了数据传输而获得本地总线的控制权，使用的是主CPU的$\overline{\text{RQ/GT}}$信号。

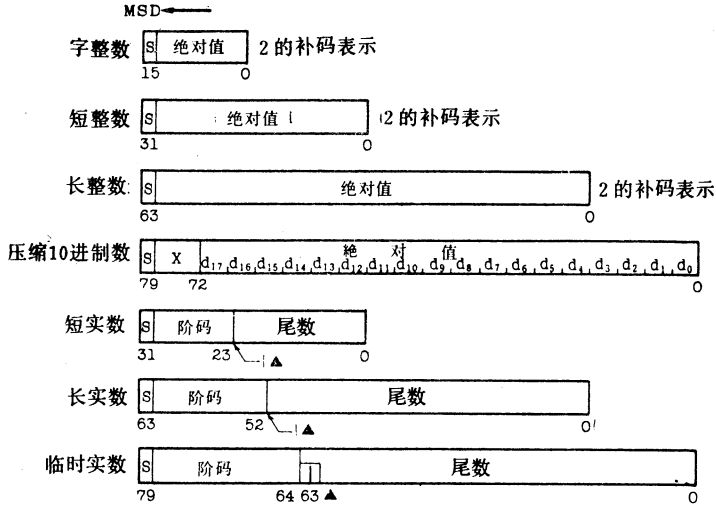
8087处理数据的类型和数值范围如表9.2所示，数据的格式如图9.3所示。定点二进制数可以处理到64位，十进制数可以处理到18位

表9.1 8087与8086软件运算速度的比较

指令种类	执行时间〔μS〕（近似值）（5MHz时钟）	
	8087	8086软件仿真
乘法（单精度）	19	1600
乘法（双精度）	27	2100
加法	17	1600
除法（单精度）	39	3200
比较	9	1300
装入（单精度）	9	1700
存储（单精度）	18	1200
开平方	36	19600
正 切	90	13000
指数运算	100	17100

表9.2 8087的数据类型及其范围

数据类型	引脚数	有效位数（十进制）	概略范围（十进制）
字整数	16	4	$-32768 \leqq x \leqq +32767$
短整数	32	9	$-2\times10^{9} \leqq x \leqq +2\times10^{9}$
长整数	64	18	$-9\times10^{18} \leqq x \leqq +9\times10^{18}$
压缩型十进制	80	18	$-99\cdots99 \leqq x \leqq +99\cdots99$（18位数）
短实数	32	6—7	$8.43\times10^{-37} \leqq \lvert x\rvert \leqq 3.37\times10^{38}$
长实数	64	15—16	$4.19\times10^{-307} \leqq \lvert x\rvert \leqq 1.67\times10^{308}$
暂存实数值	80	19	$3.4\times10^{-4932} \leqq \lvert x\rvert \leqq 1.2\times10^{4932}$



〔注〕 1. S：符号（0 = 正，1 = 负）
2. d_n：10进制数（每字节为2位10进制数）
3. X：无任何意义
4. ▲：二进制小数点隐含在内
5. |：暂时为实数时使用指数部分的整数位

图9.3 8087的数据形式

由于8087浮点运算的格式附合IEEE正在制定中的小型机/微型机的工业标准，因此它可以促进与其他计算机之间数据运算程序的可移植性。

因为8087具有能进行18位十进制运算、浮点运算、三角函数及指数运算等很强的运算能力，所以能广泛地应用在事务数据处理、过程控制、数值控制、机器人、导航、图形显示终端及数据收集等方面。

表9.3 8087指令一览表

	加　　法
FADD	实数加法
FADDP	实数加法及弹出堆栈
FIADD	整数加法
	减　　法
FSUB	实数减法
FSUBP	实数减法及弹出堆栈
FISUB	整数减法
FSUBR	实数反减法
FSUBRP	实数反减法及弹出堆栈
FISUBR	整数反减法
	乘　　法
FMUL	实数乘法
FMULP	实数乘法及弹出堆栈
FIMUL	整数乘法
	除　　法

FDIV	实数除法
FDIVP	实数除法及弹出堆栈
FIDIV	整数除法
FDIVR	实数反除法
FDIVRP	实数反除法及弹出堆栈
FIDIVR	整数反除法

	其　他　指　令
FSQRT	开平方
FSCALE	比　例
FPREM	部分剩余
FRNDINT	求　整
FXTRACT	分出指数及尾数
FABS	绝对值
FCHS	符号反转

9.2 高速 I/O处理器8089

随着微型计算机系统的发展，如果仍像以前的处理器那样，包括I/O控制在内的所有工作均由一个CPU承担，则CPU的时间几乎全被I/O操作所占有，从而降低了系统的吞吐量。同时，最近与这些系统连接的外部设备也要求高速的数据传输，特别是在实时处理时，要求实时服务与主CPU并行处理。

8089 I/O处理器与8086配合就可以解决上述问题。它采用了在大型机上使用的智能I/O子系统，和通道控制器等设计思想，很适用于以8086CPU为中心的微型计算机系统的领域。

在使用8089的系统中，将外围I/O从CPU分离出来，只在需要时才通过CPU，从而大大提高了系统的吞吐量。表9.4表示数据

传输速率。

表9.4 8089的数据传输速率

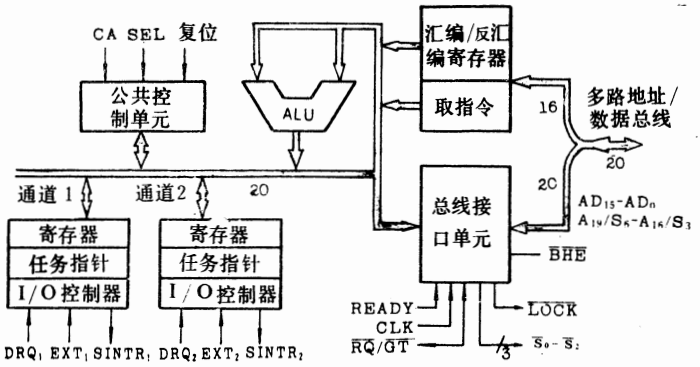
	本地总线		远程	
	字节	字	字节	字
带宽〔千字节/S〕	830	1250	830	1250
等待时间〔μS〕	1.0/2.4*	1.0/2.4*	1.0/2.4*	1.0/2.4*
系统总线使用时间〔μS〕	传送一次2.4	传送一次1.6	传送一次0.8	传送一次0.8

* 在通道处于请求等待状态时为1.0μS，其他情况以及与其他通道交叉时为2.4μS

图9.4表示使用8089的组成情况，从图中可以看出它有两个独立的I/O通道及其专用的寄存器组和指令指针，这两个通道可以独立地执行DMA传输和一连串的指令。8089的寄存器如图9.5所示，它有两组完全相同的寄存器组。GA、GB都是20位，可以指定系统总线或本地总线。而且在DMA传输时，GA、GB用于指示源地址和目标地址，并可自动增量。GC寄存器用于代码变换（如ASCII⇄EBCDIC等）等翻译操作，在DMA传输时作为查表指针。

处于传输中的每个数据位的操作，都可以测试和屏蔽。CPU和IOP之间的通信可以通过通道注意信号（CA）和中断线进行，任务程序、状态信息及参数等可通过存储器模块相互交换。

DMA操作不仅在通常的存储器和I/O间进行，而且也可以扩充到存储器和存储器间及I/O和I/O间进行，从而能够实现高速模块传输。



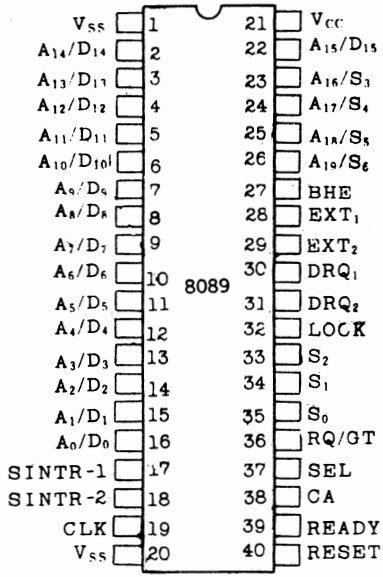


图9.4 8089 I/O处理器方框图和引脚排列图

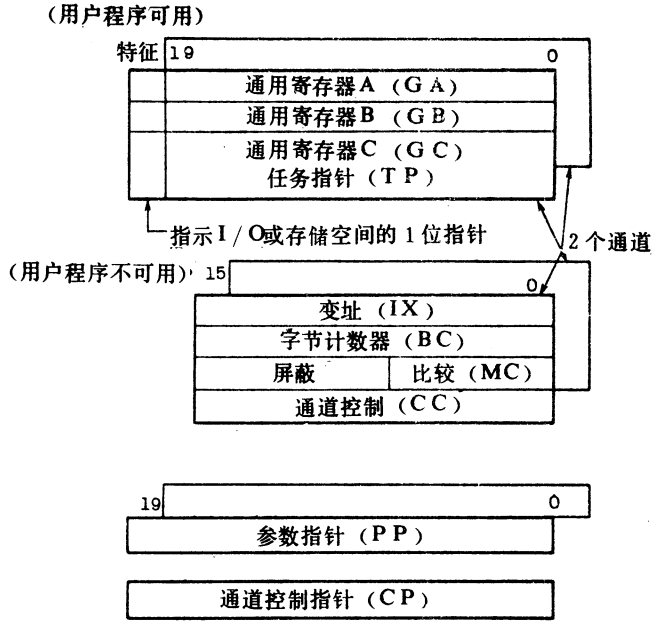


图9.5 8089寄存器的组成

9.3 8086未来的发展方向

以8086为中心，配上高速运算芯片(8087)及I/O处理器（8089）所组成的系统与8位系统比较，处理能力可提高一个数量级，实际上这是16位以上微型计算机的开始，下一步的工作是随着系统的发展，为减少软件开发成本，应采取图9.7所示的步骤，即把软件直接作在芯片上。

第一步把操作系统的核心部分作在芯片上，第二步再把高级语

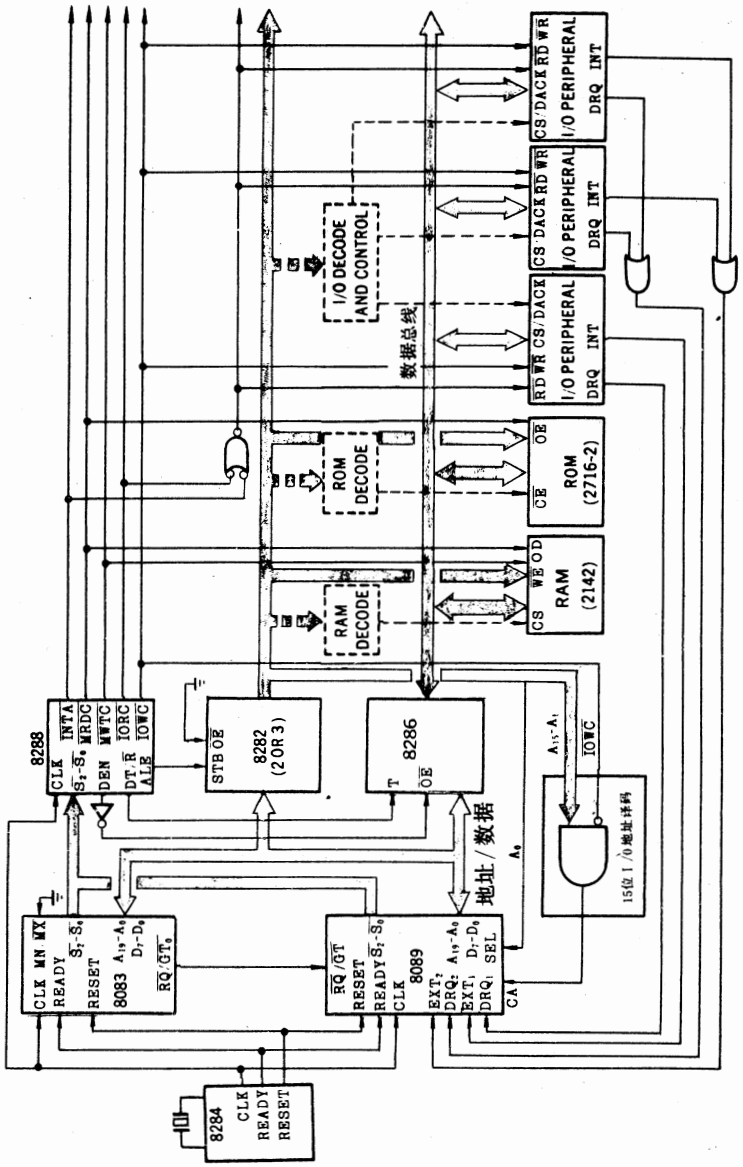


图9.6　8088/8089的本地方式组成图

言以固件形式作在芯片上。下面简单介绍INTEL公司在8086以后发表的三种CPU。表9.5为这些CPU的一览表。

（1）iAPX186/188（Micro Midi）（16位CPU）

iAPX186/188把8086/8088系统中的振荡器等外接电路，和操作系统的核心部分与CPU装在同一芯片上，使系统价格降低及向上兼容。

（2）iAPX286（Micro Maxi）（16/32位CPU）

iAPX286与8086的程序代码具有互换性，因此增强了功能。APX286的方框图如图9.8所示。

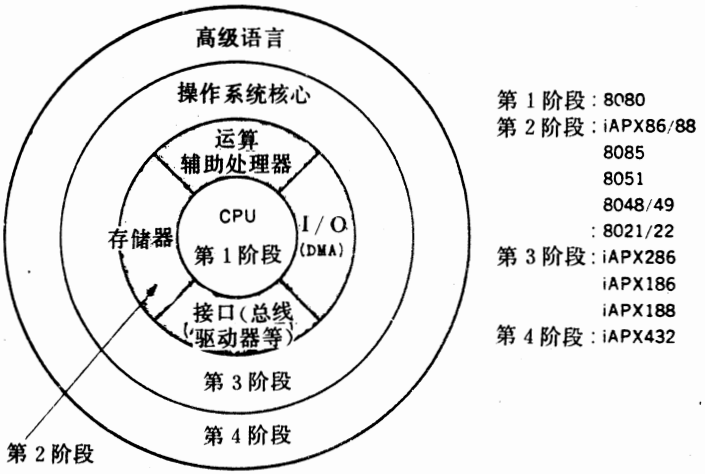


图9.7　微型计算机系统80的层次结构

表9.5 微型计算机系统的分类

微型计算机系统级别名	功能	相对性能		存储器		
		CPU	I/O	程序大小	存储器管理	代表产品
微型主机	32位	20～70	2～15	256k～8M		iAPX432
最大微型	16/32位	25	4	128k～1M		iAPX286
中等微型	16位	8～10	2	32k～256k		iAPX86
微型计算机	8/16位	1～5	1～1.5	16k～120k		iAPX88
微型控制器	8位	1	0.3	4k～32k		8048
						8051
						8085

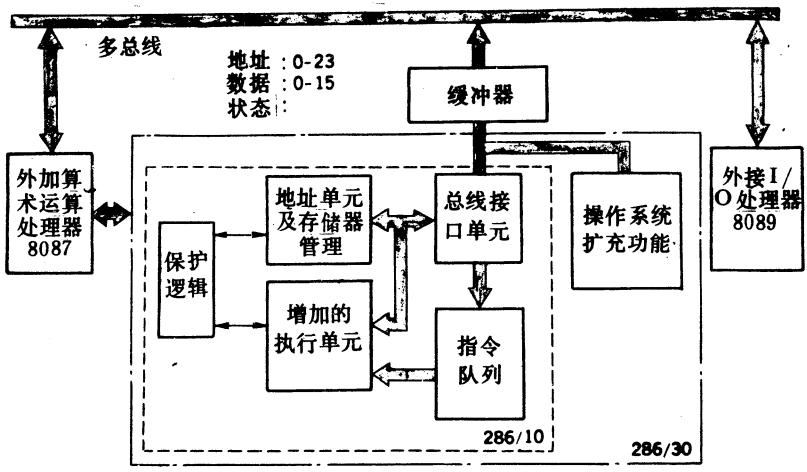


图9.8 iAPX286/10及iAPX286/30

iAPX286具有下述功能：

- 物理地址空间扩充到16Mb。
- 每一任务有1Gb的虚拟存储空间的管理功能。

• 采用了流水线结构，具有5倍于标准8086的处理速度。

• 在芯片上具有存储器管理和保护机构及多级软件保护功能。

• 具有8086的扩充指令系统，芯片上装有操作系统的核心部分。

（3）iAPX432（Micro Main frame）（32位CPU）

iAPX432是32位的CPU，它在微型机系统中采用了大型主机（Main frame）的功能。图9.9为iAPX432与8086系列及其多总线间的接口处理器、用于分散处理数据的数据处理器及存储器三个芯片构成的系统。此外，通过高级语言和操作系统的功能作在芯片上，可以减少软件开发的成本。

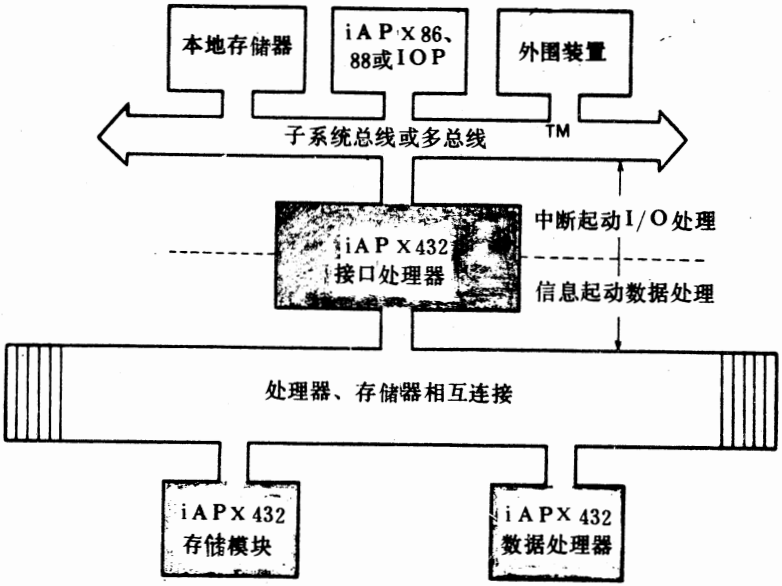


图9.9 iAPX432系统的组成

附录 1　8086 指令详解

（按英文字母顺序排列）

AAA

(ASCII adjust for addition)——加法的ASCII修正指令

操作： 若AL的低4位大于9或辅助进位标志AF被置为“1”，则往AL中加6，往AH中加1，AF与CF标志均被置位。结果AL中新的数值的高4位应为“0”，而低4位是上述加法所得出的数（0～9之间的数）。

```
if ((AL) & 0FH) > 9 or (AF) = 1 then
  (AL) ← (AL) + 6
  (AH) ← (AH) + 1
  (AF) ← 1
  (CF) ← (AF)
  (AL) ← (AL) & 0FH
```

编码：

0 0 1 1 0 1 1 1

定时(时钟)： 4

实例：　AAA　；在加法指令的后面使用

标志： 影响　AF, CF.
不确定　OF, PF, SF, ZF

说明：AAA 指令用来对两个非压缩型的 BCD (ASCII) 数相加的结果（在 AL 中）进行修正，以获得一个非压缩型的十进制“和”。

AAD

(ASCII adjust for division)——除法的ASCII修正指令

操作：累加器的高位字节 (AH) 乘10，然后与低位字节 (AL) 相加，结果存于AL中，再把AH清 0 。

(AL) ← (AH)*OAH + (AL)
(AH) ← 0

编码：

11010101	00001010

定时(时钟)：60

实例：　AAD ；在除法指令的前面使用

标志：影响　PF, SF, ZF.
不确定　AF, CF, OF

说明：在两个非压缩型十进制数相除指令之前，用AAD指令对AL中的被除数进行修正，使除法指令执行后获得的商是一个非压缩型的十进制数。

AAM

(ASCII adjust for multiply) ——乘法的ASCII修正指令

操作： 用AL除10的结果去置换AH中的内容，然后再用这次除法所产生的余数去代替AL中的内容，也就是对AL取模10的余数去代替AL的内容。

(AH) ← (AL) / 0AH
(AL) ← (AL) % 0AH

编码：

11010100	00001010

定时(时钟)： 83

实例： AAM ;在乘法指令后面使用

标志： 影响 PF, SF, ZF.
不确定 AF, CF, OF

说明： AAM指令的功能是对两个非压缩型十进制数相乘的结果(在AX中）进行修正，以获得非压缩型的十进制积。

AAS

(ASCII adjust for subtraction)——减法的ASCII修正指令

操作： 若AL的低4位大于9，或辅助进位标志AF置位，则从AL中减去6，从AH中减去1，AF和CF标志被置位。AL中老

的数值被指令所建立的新数值代替，这个新数值的高4位为0，低4位是0～9中的一个数。

```
if ((AL) & 0FH) > 9 or (AF) = 1 then
  (AL) ← (AL)−6
  (AH) ← (AH)−1
  (AF) ← 1
  (CF) ← (AF)
  (AL) ← (AL) & 0FH
```

编码：

00111111

定时(时钟)： 4

实例： AAS ；在减法指令后面使用

标志： 影响 AF, CF.
不确定 OF, PF, SF, ZF

说明： AAS 指令对两个非压缩型十进制数相减的结果（在AL中）进行修正，以获得非压缩型十进制数的差值。

ADC
(Add with carry)—— 带进位的加法指令

操作： 如果进位标志CF已经置位，那么ADC指令在往目的操作数（最左边）存入结果之前，要先将结果加1。若进位标志CF没有置位（即为0），则不往结果中加1。

if (CF) = 1 then (DEST) ← (LSRC) + (RSRC) + 1
else (DEST) ← (LSRC) + (RSRC)

编码：有三种格式

存储器或寄存器操作数与寄存器操作数相加：

0 0 0 1 0 0 d w	mod reg r/m

if d = 1 then LSRC = REG, RSRC = EA, DEST = REG
else LSRC = EA, RSRC = REG, DEST = EA

定时(时钟)：

(a) 寄存器到寄存器	3
(b) 存储器到寄存器	9+EA
(c) 寄存器到存储器	16+EA

实例：

```
(a) ADC   AX,   SI
    ADC   ,SI   ;和上面的功能一样
    ADC   DI,   BX
    ADC   CH,   BL

(b) ADC   DX,   MEM_WORD
    ADC   AX,   BETA [SI]
    ADC   ,BETA [SI]  ;与上面一样
    ADC   CX,   ALPHA [BX] [SI]

(c) ADC   BETA [DI],   BX
    ADC   ALPHA [BX] [SI],   DI
    ADC   MEM_WORD,   AX
```

立即操作数到累加器：

0 0 0 1 0 1 0 w	data	data if w=1

if w = 0 then LSRC = AL, RSRC = data, DEST = AL
else LSRC = AX, RSRC = data, DEST = AX

定时(时钟)： 4

实例：

```
ADC   AL,  3
ADC   AL,  VALUE_13_IMM
ADC   AX,  333
ADC   AX, IMM_VAL_777
ADC   ,IMM_VAL_777 ;与上面的指令一样
```

立即操作数到存储器/寄存器操作数：

1 0 0 0 0 0 s w	mod 0 1 1 r/m	data	data if s:w=01

LSRC = EA, RSRC = data, DEST = EA

定时(时钟)： (a) 立即数到存储器 17 + EA
(b) 立即数到寄存器 4

实例：

```
(a) ADC   BETA [SI],  4
    ADC   ALPHA [BX] [DI],  IMM4
    ADC   MEM_LOC,  7396

(b) ADC   BX,  IMM_VAL_987
    ADC   DH,  65
    ADC   CX,  432
```

如果寄存器或存储器的字要与立即数字节相加，那么在加之前，要先进行符号扩展，把符号扩展到最高位（第16位），即把字节立即数扩展为字立即数。

标志： 影响　AF, CF, OF, PF, SF, ZF

说明： ADC 指令实现两个操作数的相加，若CF置位，则将结果加1后，送回目的操作数中去。

ADD

(Addition) ——加法指令

操作： 将两个操作数的和存入目的（左边的）操作数中去。

(DEST) ← (LSRC) + (RSRC)

编码： 有三种格式

存储器/寄存器操作数与寄存器操作数：

0 0 0 0 0 0 d w	mod reg r/m

if d = 0 then LSRC = REG, RSRC = EA, DEST = REG
else LSRC = EA, RSRC = REG, DEST = EA

定时(时钟)：

(a) 寄存器到寄存器	3
(b) 存储器到寄存器	9+EA
(c) 寄存器到存储器	16+EA

实例：

```
(a) ADD   AX,   BX
    ADD   ,BX   ;与上面一样
    ADD   CX,   DX
    ADD   DI,   SI
    ADD   BX,   BP

(b) ADD   CX,   MEM_WORD
    ADD   AX,   BETA [SI]
    ADD   ,BETA [SI] ;与上面一样
    ADD   DX,   ALPHA [BX] [DI]

(c) ADD   GAMMA  [BP] [DI],   BX
    ADD   BETA   [DI],   AX
    ADD   MEM_WORD,   CX
    ADD   MEM_BYTE,   BH
```

立即操作数到累加器：

0 0 0 0 0 1 0 w	data	data if w=1

if w = 0 then LSRC = AL, RSRC = data, DEST = AL
else LSRC = AX, RSRC = data, DEST = AX

定时(时钟)： 4

实例：

```
ADD   AL,   3
ADD   AX,   456
ADD   AL,   IMM_VAL_12
ADD   AX,   IMM_VAL_8529
ADD   ,IMM_VAL_6AB9H ;目的操作数为AX
```

立即操作数到存储器/寄存器操作数：

1 0 0 0 0 0 s w	mod 0 0 0 r/m	data	data if s:w=01

LSRC = EA, RSRC = data, DEST = EA

定时(时钟)： (a) 立即数到存储器　　17+EA
(b) 立即数到寄存器　　4

实例：

```
(a) ADD   MEM_WORD,  48
    ADD   GAMMA [DI],  IMM_84
    ADD   DELTA [BX] [SI],  IMM_SENSOR_5

(b) ADD   BX,  ORIG_VAL
    ADD   CX,  STANDARD_COUNT
    ADD   DX,  1776
```

如果寄存器或存储器的字要与立即数字节相加，那么在相加以前，要先进行符号扩展，将字节立即数扩展为字立即数(16位的)。

标志： 影响　AF, CF, OF, PF, SF, ZF

说明： ADD 指令使两个操作数相加并将结果送回目的（左边的）操作数中去。

AND

(And: logical conjunction) —— 与（逻辑乘法）指令

操作：将两个操作数相“与”，只有在两个操作数中对应位置上均为“1”的那些位的结果才为“1”，其它情况的那些位的结果为“0”。结果存入目的（左边的）操作数中去，进位和溢出标志被复位为“0”。

(DEST) ← (LSRC) & (RSRC)
(CF) ← 0
(OF) ← 0

编码：有三种格式

存储器/寄存器操作数与寄存器操作数：

0 0 1 0 0 0 d w	mod reg r/m

if d = 1 then LSRC = REG, RSRC = EA, DEST = REG
else LSRC = EA, RSRC = REG, DEST = EA

定时(时钟)：

寄存器到寄存器	3
存储器到寄存器	9+EA
寄存器到存储器	16+EA

实例：

```
(a) AND   AX,   BX
    AND   ,BX   ;与上面一样
    AND   CX,   DI
    AND   BH,   CL

(b) AND   SI,   MEM_NAME_WORD
    AND   DX,   BETA [BX]
    AND   BX,   GAMMA [BX] [SI]
    AND   AX,   ALPHA [DI]
    AND   ,ALPHA [DI]  ;与上面一样
    AND   DH,   MEM_BYTE
```

```
(c) AND   MEM_NAME_WORD,   BP
    AND   ALPHA [DI],   AX
    AND   GAMMA [BX] [DI],   SI
    AND   MEM_BYTE,   AL
```

立即操作数到累加器：

0 0 1 0 0 1 0 w	data	data if w=1

if w = 0 then LSRC = AL, RSRC = data, DEST = AL
else LSRC = AX, RSRC = data, DEST = AX

定时(时钟)：立即数到寄存器　　4

实例：

```
AND   AL,_ 7AH
AND   AH,   0EH
AND   AX,   IMM_VAL_MASK 3
```

立即操作数到存储器/寄存器操作数：

1 0 0 0 0 0 0 w	mod 1 0 0 r/m	data	data if w=1

LSRC = EA, RSRC = data, DEST = EA

定时(时钟)：(a) 立即数到寄存器　　4
(b) 立即数到存储器　　17 + EA

实例：

```
(a) AND   BL,   10011110B
    AND   CH,   3EH
    AND   DX,   7A46H
    AND   SI,   987
```

(b) AND MEM_WORD, 7A46H
AND MEM_BYTE, 46H
AND GAMMA [DI], IMM_MASK 14
AND CHI_BYTE [BX] [SI], 11100111B

标志： 影响 CF, OF, PF, SF, ZF.
不确定 AF

说明： AND 指令对两个操作数按位进行逻辑“与”，结果送回目的（左边的）操作数中去。如果原来两个操作数中的对应位为“1”，则结果位为“1”；否则结果位为“0”。

CALL

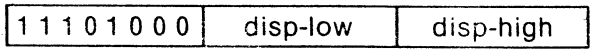
(Call a procedure)——调用（一个过程）指令

操作： 如果是段间的调用，堆栈指示器减 2，并将 CS 寄存器的内容保存进栈，再将双字段间指示器中的第二个字（段值）填入 CS。然后堆栈指示器再减 2，并将指令指示器的内容保存进栈，最后的一步是用目标操作数的偏移值（DEST）取代 IP 中的内容。目标操作数的偏移值就是被调用“过程”的第一条指令的偏移地址。段内或群内调用只作第 2）、3）、和 4）步。

1) 如果是段间调用，那么执行
(SP) ← (SP)−2
((SP) + 1:(SP)) ← (CS)
(CS) ← SEG
2) (SP) ← (SP)−2
3) ((SP) + 1:(SP)) ← (IP)
4) (IP) ← DEST

编码：

直接段内或群内调用：



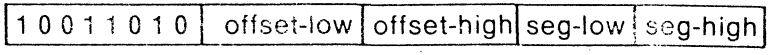
DEST = (IP) + disp

定时(时钟)： 13 + EA

实例：

```
CALL  NEAR_LABEL
CALL  NEAR_PROC
```

段间直接调用：



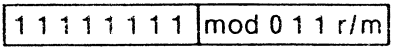
DEST = offset, SEG = seg

定时(时钟)： 20

实例：

```
CALL  FAR_LABEL
CALL  FAR_PROC
```

段间间接调用：



DEST = (EA), SEG = (EA + 2)

定时(时钟)：29 + EA

实例：

```
CALL  DWORD  PTR  [BX]
CALL  DWORD  PTR  VARIABLE_NAME [SI]
CALL  MEM_DOUBLE_WORD
```

间接段内或群内调用：

1 1 1 1 1 1 1 1	mod 0 1 0 r/m

DEST = (EA)

定时(时钟)：11

实例：

```
CALL  WORD  PTR  [BX]
CALL  WORD  PTR  VARIABLE_NAME
CALL  WORD  PTR  [BX] [SI]
CALL  WORD  PTR  [DI]
CALL  WORD  PTR  VARIABLE_NAME [BP] [SI]
CALL  MEM_WORD
CALL  BX
CALL  CX
```

标志：不影响

说明：CALL指令将下条指令的偏移地址保存进栈(若为段间调用，则CS 寄存器的内容先进栈)，然后将控制转给目标操作数。直接调用和转移只能使用相对于CS的标号，而不能用变量。如果在指令中，或目标标号的说明中，没有指明 是 FAR 属性，就假定是NEAR 属性。

在上面间接调用的实例中，通过变量的调用，可以利用PTR操作符来指出所要用的一个字（对NEAR调用），或两个字（对FAR标号或过程的调用）。利用字寄存器进行间接调用是属于NEAR调用。如果在CALL指令中沒有加段修改前缀，或者利用BP，在寄存器间接调用中所用的隐含段寄存器就是DS。隐含的段寄存器用于构成包括调用目标的偏移值地址（如果是“长”调用，则还包括段的地址）。如果CALL指令中用BP 则隐含的段寄存器是SS。然而，如果指明了段的前缀，例如：

```
CALL  WORD  PTR  ES: [BP] [DI]
```

则段寄存器是利用ES。

对于通过变量或地址表达式的间接调用指令，隐含的段寄存器由源程序行中的地址表达式，和所用的伪指令.ASSUME来确定。

当CALL指令用于转移控制时，RET是隐含的。在间接调用的情况下，必需注意保证CALL的类型和RET的类型一致，否则就会出错，而且难于发现，关键在于CS是否能被保存和恢复。

CBW

(Convert byte to word)——变换字节为字的指令

操作： 如果累加器（AX）的低位字节（AL）的内容小于80H，则累加器的高位字节（AH）为00H；否则，若累加器（AX）的低位字节（AL）中的内容≥80H，则累加器的高位字节（AH）为FFH。也就是说AH中各位的值与AL中最高位（位7）的值一样。

编码：

1 0 0 1 1 0 0 0

定时(时钟)： 2

实例： CBW

标志： 不影响

说明： CBW 指令把AL中的符号位扩展到AH 中各个位。这条指令通常用来由字节产生双倍长（字）的除数，因此要把 CBW 指令放在除法指令前面。

CLC

(Clear carry flag)——清进位标志指令

操作： 将进位标志清 0 。

(CF) ← 0

编码：

1 1 1 1 1 0 0 0

定时(时钟)： 2

实例： CLC

标志： 影响 CF

说明： CLC 指令用来将CF标志清 0 。

CLD

(Clear direction flag)——清除方向标志指令

操作： 将方向标志清 0 。

(DF) ← 0

编码：

1 1 1 1 1 1 0 0

定时(时钟)： 2

实例： CLD

标志： 影响 DF.

说明： CLD 指令用来将DF 标志清 0，使字符串操作中的“操作数指示器”（SI 和/或 DI）自动增量。

CLI

(Clear interrupt flag)——清除中断标志指令

操作： 将中断标志清 0 。

(IF) ← 0

编码：

1 1 1 1 1 0 1 0

定时(时钟)： 2

实例： CLI

标志： 影响 IF

说明： 清除IF标志，从而屏蔽了出现在8086 INTR 线上的“可屏蔽外部中断”（在8086 NMI 线上的非屏蔽中断不能禁止）。

CMC

(Complement carry flag)——进位标志求反指令

操作： 若进位标志CF为0，则将CF置1；

if (CF) = 0 then (CF) ← 1 else (CF) ← 0

编码：

1 1 1 1 0 1 0 1

定时(时钟)： 2

实例： CMC

标志： 影响 CF

说明： CMC将CF标志取反。

CMP

(Compare two operands) ——比较两个操作数指令

操作：从目的操作数（左边的）中减去源操作数（右边的）。按减的结果影响标志位，但结果不送回，即保留原操作数不变。

(LSRC)-(RSRC)

编码：有三种格式

存储器/寄存器操作数与寄存器操作数比较：

0 0 1 1 1 0 d w	mod reg r/m

if d = 1 then LSRC = REG, RSRC = EA
else LSRC = EA, RSRC = REG

定时(时钟)：

(a) 寄存器与寄存器	3
(b) 存储器与寄存器	9+EA
(c) 寄存器与存储器	9+EA

实例：

```
(a) CMP   AX,   DX
    CMP   ,DX ;和上面一样
    CMP   SI,   BP
    CMP   BH,   CL

(b) CMP   MEM_WORD,   SI
    CMP   MEM_BYTE,   CH
    CMP   ALPHA [DI],   DX
    CMP   BETA [BX] [SI],   CX

(c) CMP   DI,   MEM_WORD
    CMP   CH,   MEM_BYTE
    CMP   AX,   GAMMA [BP] [SI]
```

立即操作数与累加器：

0 0 1 1 1 1 0 w	data	data if w=1

if w = 0 then LSRC = AL, RSRC = data
else LSRC = AX, RSRC = data

定时(时钟)：立即数与寄存器　　4

实例：

```
CMP   AL,  6
CMP   AL,  IMM_VALUE_DRIVE 11
CMP   AX,  IMM_VAL_909
CMP   ,999
CMP   AX,  999 ；与上面一样
```

立即操作数与存储器/寄存器操作数：

1 0 0 0 0 0 s w	mod 1 1 1 r/m	data	data if s:w=01

LSRC = EA, RSRC = data

定时(时钟)：(a) 立即数与寄存器　　4
(b) 立即数与存储器　　17+EA

实例：

```
(a) CMP   BH,  7
    CMP   CL,  19_IMM_BYTE
    CMP   DX,  IMM_DATA_WORD
    CMP   SI,  798

(b) CMP   MEM_WORD,  IMM_DATA_BYTE
    CMP   GAMMA [BX],  IMM_BYTE
    CMP   [BX] [DI],  6ACEH
```

如果把一个立即数字节与一个寄存器/存储器字进行比较，那么在比较之前，先要对立即数字进行符号扩展，使其成为16位的字。这时指令操作码字节为83H（即S：W位均为1）。

标志： 影响　　AF, CF, OF, PF, SF, ZF

说明： CMP指令执行减法操作，只影响标志，结果不回送。通常源操作数（右边的）与目的操作数（左边的）的类型（字或字节）要相同。只有立即数字节与存储器字的比较是例外。

CMPS

(Compare byte string, compare word string)——比较字节串，比较字串指令

操作： 源操作数减去目的操作数。源操作数用SI作为变址寄存器，目的操作数用DI作为指向附加段的变址寄存器。但应注意，这条指令的源操作数写在左边，而目的操作数写在右边，这是把DI变址的操作数作为右边操作数的唯一的一条字符串指令。这条指令影响标志，但不影响操作数本身。如果DF为0，则SI和DI都是增量的；如果DF为1，则SI和DI都减量，增/减量之后就指向所比较的字符串的下一单元，对字节串为加1或减1；对字串为加2或减2。

```
(LSRC)-(RSRC)
if (DF) = 0 then
  (SI) ← (SI) + DELTA
  (DI) ← (DI) + DELTA
else
  (SI) ← (SI)-DELTA
  (DI) ← (DI)-DELTA
```

编码：

1 0 1 0 0 1 1 w

if w = 0 then LSRC = (SI), RSRC = (DI), DELTA = 1 (BYTE)
else LSRC = (SI) + 1:(SI), RSRC = (DI) + 1:(DI), DELTA = 2 (WORD)

定时(时钟)： 22

实例：

```
MOV  SI,  OFFSET  STRING1
MOV  DI,  OFFSET  STRING2
CMPS  STRING 1,  STRING2
;在 CMPS 指令中命名的“名字操作数”只是对汇编程序来
;说的。汇编程序用它们来识别字的类型及当前段寄存器内容的可
;访向性。实际上 CMPS 指令只用 SI 和 DI 去指示要进行内容比较
;的单元，而不用在源程序行中给出名字。
```

标志： 影响　　AF, CF, OF, PF, SF, ZF

说明： CMPS 指令使SI 寻址的（字节/字）操作数中减去DI 寻址（字节/字）操作数，该指令只影响标志，结果不回送。

CMPS 指令重复操作时，能对两个字符串进行比较。采用适当的重复前缀就能比较两个字符串。一旦发现有两个元素不等时，就结束比较，从而建立字符串之间的顺序。

注意：由DI 变址指定的操作数是指令中右边的操作数，并且这个操作数只能用ES 寄存器来寻址—这个缺省的段寄存器是不能修改的。

CWD

(Convert word to doubleword) ——变换字为双字指令

操作： 用AX的最高位去替换DX中的各个位。

```
if (AX) < 8000H then (DX) ← 0
else (DX) ← FFFFH
```

编码：

1 0 0 1 1 0 0 1

定时(时钟)： 5

实例： CWD

标志： 不影响

说明： CWD指令用来将AX寄存器内容的符号扩展到DX中去。

DAA

(Decimal adjust for addition) ——加法的十进制修正指令

操作： 若AL寄存器的低4位大于9或辅助进位标志已经置位，则加6到AL中并将AF置位；若AL寄存器内容大于9FH或者进位标志置位，则加60H到AL中去并将CF置位。

```
if (AL) & 0FH) > 9 or (AF) = 1 then
  (AL) ← (AL) + 6
  (AF) ← 1
if (AL) > 9FH or (CF) = 1 then
  (AL) ← (AL) + 60H
  (CF) ← 1
```

编码：

0 0 1 0 0 1 1 1

定时(时钟)： 4

实例： DAA

标志： 影响 AF, CF, PF, SF, ZF
不确定 OF

说明： DAA指令用来对两个压缩型十进制数相加的结果（在AL中）进行修正和产生一个压缩型的十进制“和”。

DAS

(Decimal adjust for subtraction)—— 减法的十进制修正指令

操作： 若AL的低4位大于9或AF为1，则AL减去6并将AF置1；若AL大于9FH或CF为1，则AL减去60H并将CF置位。

```
if (AL) & 0FH) > 9 or (AF) = 1 then
   (AL) ← (AL)−6
   (AF) ← 1
if (AL) > 9FH or (CF) = 1 then
   (AL) ← (AL)−60H
   (CF) ← 1
```

编码：

0 0 1 0 1 1 1 1

定时(时钟)： 4

实例： DAS

标志： 影响 AF, CF, PF, SF, ZF.
不确定 OF

说明： DAS指令对两个压缩型十进制数相减的结果（在AL中）进行修正，以获得压缩型的十进制结果。

DEC

(Decrement destination by one)——将“目的”减“1”指令

操作： 将指定的操作数减1。

(DEST) ← (DEST)−1

编码： 有两种格式

寄存器操作数（字）：

0 1 0 0 1 reg

DEST = REG

定时(时钟)： 2

实例：

```
DEC  AX
DEC  DI
DEC  SI
```

存储器/寄存器操作数：

1 1 1 1 1 1 1 w	mod 0 0 1 r/m

DEST = EA

定时(时钟)：寄存器　　2
存储器　　15+EA

实例：

```
DEC  MEM_BYTE
DEC  MEM_BYTE [DI]
DEC  MEM_WORD
DEC  ALPHA [BX] [SI]
DEC  BL
DEC  CH
```

标志：影响　AF, OF, PF, SF, ZF
不确定　CF

说明：DEC（减 1）指令用来使操作数减 1，并将结果送回该操作数。

DIV

(Division, unsigned)——无符号数的除法指令

操作：如果除法产生的结果（商的值）超出了保存它的目的寄存器的范围，就如同被 0 除了一样，产生 0 型的中断，标志被保存进栈，IF 和 TF 被复位为 0，CS 寄存器的内容保存进栈，然后用存放在内存单元 2 和 3 中的字填入 CS，IP 寄存器保存进栈，然后用放在内存单元 0 和 1 中的字填入 IP。以上

操作序列包括了一次“长调用”，它用来调用0类型中断的处理“过程”（服务程序），此“过程”的入口段地址和偏移地址分别存放在内存的单元2、3和单元0、1中。

如果除法的结果（商）未超过存放它的寄存器的范围，则商存在AL或AX(对字操作数)中，余数存放在AH或DX中。

```
(temp) ← (NUMR)
if (temp) / (DIVR) > MAX 执行下面的操作序列
        (QUO), (REM)  不确定
        (SP) ← (SP)-2
        ((SP)+1:(SP)) ← FLAGS
        (IF) ← 0
        (TF) ← 0
        (SP) ← (SP)-2
        ((SP)+1:(SP)) ← (CS)
        (CS) ← (2)  i.e.  内存单元2和3的内容
        (SP) ← (SP)-2
        ((SP)+1:(SP)) ← (IP)
        (IP) ←  (0) i.e., 内存单元0和1的内容
else
        (QUO) ← (temp) / (DIVR),   此处 / 为无符号除法
        (REM) ← (temp) % (DIVR)    此处 % 为取无符号的模
```

编码：

1 1 1 1 0 1 1 w	mod 1 1 0 r/m

(a) if w = 0 then NUMR = AX, DIVR = EA, QUO = AL, REM= AH, MAX = FFH

(b) else NUMR = DX:AX, DIVR = EA, QUO = AX, REM = DX, MAX = FFFFH

定时(时钟)： 8位运算　90+EA

16位运算　155+EA

实例:

(a1) 字被字节除

```
MOV  AX,  NUMERATOR_WORD
DIV  DIVISOR_BYTE
; 商在 AL 中，余数在 AH 中
```

(a2) 字节被字节除

```
MOV  AL,  NUMERATOR_BYTE
CBW    ; 变换 AL 中的字节为 AX 中的字
DIV  DIVISOR_BYTE
;商在 AL 中，余数在 AH 中
```

(b1) 双字被字除

```
MOV  DX,  NUMERATOR_HI_WORD
MOV  AX,  NUMERATOR_LO_WORD
DIV  DIVISOR_WORD
; 商在 AX 中，余数在 DX 中
```

(b2) 字被字除

```
MOV  AX,  NUMERATOR_WORD
CWD    ; 将字变换为双字
DIV  DIVISOR_WORD
; 商在 AX 中，余数在 DX 中
```

注释: 上面实例中的每个存储器操作数可以是任何变量或有效的地址表达式，只要它们的类型是一样的就行了。例如，在上面(a1)中的NUMERATOR_WORD可用表达式ARRAY_NAME [BX] [SI] + 67 来替换，只要ARRAY_NAME的类型为WORD.(字)。同样地，DIVISOR_BYTE能用RATE_TABLE [BP] [DI] 替换，只要RATE_TABLE 的类型为BYTE.(字节)。

标志：无有效的标志结果，即

不确定　AF, CF, OF, PF, SF, ZF

说明：DIV 指令执行无符号的除法运算，即累加器及其扩展寄存器中的双倍长度的被除数NUMR 被源操作数中的除数 DIVR所除。这里，对 8 位的无符号除法，被除数 NUMR在AL和AH中，对于16位的无符号除法，被除数NUMR在AX和DX中。

DIV 指令将单倍长度的商（QUO操作数）送回累加器（AL或AX），将单倍长度的余数（REM 操作数）送到累加器的扩展字节(对 8 位运算，累加器扩展为字节AX，对16位运算累加器扩展字节为DX)。如果商大于MAX，则商QUO和余数 REM都不确定，并产生类型0的中断，即除法错误中断。

在任何除法运算中，标志都是不确定的，所得的商若为非整数，则取整数。

ESC

(Escape)——处理器交权指令

操作：

if mod ≠ 11 then data bus ← (EA)
if mod = 11, no operation.

编码：

11011x	mod x r/m

定时（时钟）：7+EA

实例：

ESC　EXTERNAL_OPCODE,　ADDRESS；这个外部操作

; 码(EXTERNAL_OPCODE 是6位的数，它被分为两个三
; 位的字段，如编码中的X 所示。

标志：不影响

说明：ESC 指令提供了一种方法，使其它处理器能从8086指令流中接收它们的指令，并能利用8086的寻址方式。在ESC 指令期间，8086除去存取一个存储器操作数并把它放到总线上去的操作外，不作任何其它操作。

HLT

(Halt)——处理器暂停指令

操作：没有

编码：

11110100

定时（时钟）：2

实例：　　HLT

标志：不影响

说明：HLT 使8086处理器进入它的暂停状态，这个状态可用一个允许的“外部中断”或“复位”清除掉。

IDIV

(Integer division, signed) —— 带符号的整数除法指令

操作： 如果除法运算的结果，超过保存它的寄存器的范围，就会产生0类型的中断。标志进栈保存，IF和TF清0，CS寄存器的内容保存进栈，然后用单元2和单元3中的字填入CS；当前IP的内容保存进栈，然后用单元0和单元1中的字填入IP。这个操作序列包括了一个“长调用”。它调用一个中断处理过程，这个过程的段地址存储在单元2和3；偏移地址存储在0和1单元中。

若除法的结果沒有超出保存它的寄存器的范围，那么商存入AL或AX中，余数存入AH或DX中。对字节操作数为AL和AH。对字节操作数为AX和DX。

```
(temp) ← (NUMR)
if (temp) / (DIVR) > 0 and (temp) / (DIVR) > MAX
or (temp) / (DIVR) < 0 and (temp) / (DIVR) < 0−MAX−1
        then
        (QUO), (REM) 不确定
        (SP) ← (SP)−2
        ((SP)+1:(SP)) ← FLAGS
        (IF) ← 0
        (TF) ← 0
        (SP) ← (SP)−2
        ((SP)+1:(SP)) ← (CS)
        (CS) ← (2)
        (SP) ← (SP)−2
        ((SP) + 1:(SP)) ← (IP)
        (IP) ← (0)
else
        (QUO) ← (temp) / (DIVR),  此处/为带符号的除法
        (REM) ← (temp) % (DIVR)   此处%为取带符号的模
```

编码：

1 1 1 1 0 1 1 w	mod 1 1 1 r/m

(a) if w = 0 then NUMR = AX, DIVR = EA, QUO = AL, REM = AH, MAX = 7FH
(b) else NUMR = DX:AX, DIVR = EA, QUO = AX, REM = DX, MAX = 7FFFH

定时（时钟）： 8位运算 112+EA
16位运算 177+EA

实例：

```
(a) MOV   AX,   NUMERATOR_WORD [BX]
    IDIV  DIVISOR_BYTE [BX]

(b) MOV   DX,   NUM_HI_WORD
    MOV   AX,   NUM_LO_WORD
    IDIV  DIVISOR_WORD [SI]
    SEE ALSO DIV.
```

标志： 影响 AF, CF, OF, PF, SF, ZF
所有标志不确定

说明： IDIV 指令执行带符号的除法运算，累加器及其扩展寄存器中的双倍长度的有符号的被除数NUMR，被指定的源操作数中的有符号除法 DIVR 所除。这里，对于8位数的除法运算，被除数NUMR在AL和AH中；对于16位的除法运算，被除数NUMR在AX和DX中。

IDIV 指令将得到的结果：单倍长度的商（QUO操作数）送回累加器AL或AX，将单倍长度的余数（REM操作数）送回累加器的扩展寄存器AH或DX中。

如果商是正的并且大于MAX，或者商是负的并且小于(0-MAX-1)那么QUO和REM是不确定的，就象是被0除会产生类型0（除法错误）中断一样。

在任何除法中，标志总是不确定的。IDIV 指令将非整数的商截成整数，送回符号与除式分子相同的余数。

IMUL
(Integer multiply accumulator by register-or-memory; signed)——带符号的整数乘法指令

操作： 累加器（若为字节，为AL；若为字，为AX）乘以指定的操作数。若结果的高一半为结果低一半的符号扩展，则将进位标志CF和溢出标志OF都复位，否则将它们都置位。

```
(DEST) ← (LSRC) * (RSRC)  此处：* 是乘号
if (EXT) = sign-extension of (LOW) then (CF) ← 0
else (CF) ← 1;
(OF) ← (CF)
```

编码：

1 1 1 1 0 1 1 w	mod 1 0 1 r/m

(a) if w = 0 then LSRC = AL, RSRC = EA, DEST = AX. EXT = AH, LOW = AL
(b) else LSRC = AX, RSRC = EA, DEST = DX:AX, EXT = DX, LOW = AX

定时（时钟）： 8位运算 90 + EA
16位运算 144 + EA

实例：

```
(a)  MOV   AL,  LSRC_BYTE
     IMUL  RSRC_BYTE    ; 结果在AX中

(b1) MOV   AX,  LSRC_WORD
     IMUL  RSRC_WORD
     ;高8位结果在DX中 ，低8位在 AX
```

(b2) 字和字节相乘

```
MOV  AL,  MUL_BYTE
CBW ;变换AL中的字节为字，写入AX
IMUL  RSRC_WORD
;高8位结果在DX中，低8位在AX中
```

注意：上面的任何存储器操作数都可以是一个正确类型（TYPE）的变址地址表达式。例如：若ARRAY是BYTE类型，则LSRC_BYTE可以是ARRAY [SI]。若TABLE是字类型，则RSRC_WORD可以是TABLE [BX] [DI]。

标志：影响　CF, OF.
不确定　AF, PF, SF, ZF

说明：IMUL指令执行带符号的乘法运算：它将AL（或AX）累加器中的带符号数与源操作数中的带符号数相乘，把双倍长度的乘积送回累加器及其扩展器中去，对于8位的操作数，乘积在AL和AH中；对于16位的操作数，乘积在AX和DX中。

如果结果的高一半（在EXT中）是结果低一半的符号扩展，则将CF和OF清0。反之，如果结果的高一半（在EXT中）不是结果低一半的符号扩展，则将CF、OF置1。当CF和OF都置位时，它则说明AH或DX中的内容是结果的高位数字。

IN

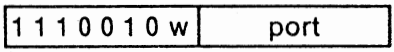
(Input byte and input word)——输入字节和输入字指令

操作：累加器的内容被指定端口的内容所替换。

(DEST)←(SRC)

编码：有两种格式

固定的端口：



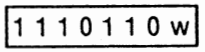
if w = 0 then SRC = port, DEST = AL
else SRC = port + 1:port, DEST = AX

定时（时钟）：10

实例：

```
IN   AX, WORD_PORT；输入字到 AX
IN   AL, BYTE_PORT ；输入一个字节到 AL
;      输入指令中的目的地必须是AX 或 AL，并且必须写出
; （即不能隐含），以便汇编程序了解输入的类型（字节还是
; 字），如上面所用到的那样。端口名字必须为 0～255 之间
; 的立即数，或者用DX寄存器的名字，但必须事先将要求的
; 端口地址送入DX中去。
```

可变的端口：



if w = 0 then SRC = (DX), DEST = AL
else SRC = (DX) + 1:(DX), DEST = AX

定时（时钟）：8

实例：

```
IN   AX, DX ；输入一个字到AX中去
IN   AL, DX ；输入一个字节到 AL中去
```

标志：不影响

说明：IN 指令从输入端口传送一个字节（或字）到AL寄存器（或AX寄存器）。端口可以用指令中的数据字节（立即数）来指定，它可指定端口 0 到端口255，或是用DX寄存器中的端口号来间接寻址，它可访问64K 个输入端口。

INC
(Increment destination by 1) —— “目的”加“1”指令

操作：指定的操作数加 1，最高位没有进位输出。

(DEST) ← (DEST) + 1

编码：有两种格式

寄存器操作数：（字）

0 1 0 0 0 reg

DEST = REG

定时（时钟）：2

实例：

```
INC   AX
INC   DI
```

存储器/寄存器操作数：

1 1 1 1 1 1 1 w	mod 0 0 0 r/m

DEST = EA

定时（时钟）：（a）寄存器　　2
（b）存储器　　15 + EA

实例：

```
(a) INC  CX
    INC  BL

(b) INC  MEM_BYTE
    INC  MEM_WORD [BX]
    INC  BYTE  PTR [Bx] ；数据段中在偏移地址 [BX] 处的字节
    INC  ALPHA [DI] [BX]
    INC  BYTE  PTR  [SI] [BP]    ；堆栈段中在偏移地址
                                 ；[SI + BP] 处的字节
    INC  WORD PTR [BX]    ；将数据段中在偏移地址为 [BX] 处
；的字加 1，于是可能有进位到“位 8”中去。
```

标志：影响　　AF, OF, PF, SF, ZF

说明：　　INC 指令将目的操作数加 1 并将结果送回目的操作数。

INT
(Interrupt) ——中断指令

操作：堆栈指示器SP 减 2，所有的标志保存进栈，然后将中断标志IF和陷阱标志TF 复位；SP 再减 2 并将CS 寄存器的当前内容保护进栈，再把双字中断矢量的高位字填到CS中去，也就是把对这个中断类型的中断处理“过程”（程序）的段基地址送入CS 寄存器；SP 再减 2，把指令指示器IP的当前内容保存进栈，然后把装配在绝对地址TYPE ★ 4 处的中断矢量的低位字送入IP。

上述操作是一个段间的“长调用”，去调用处理这种中断类型的中断服务“过程”(程序)。請参看PUSHF, INTO,, IRET.指令的说明。

(SP) ← (SP) − 2
((SP) + 1:(SP)) ← FLAGS
(IF) ← 0
(TF) ← 0
(SP) ← (SP) − 2
((SP) + 1:(SP)) ← (CS)
(CS) ← (TYPE * 4 + 2)
(SP) ← (SP) − 2
((SP) + 1:(SP)) ← (IP)
(IP) ← (TYPE * 4)

编码：

1 1 0 0 1 1 0 v	type if v=1

(a) if v = 0 then TYPE = 3
(b) else TYPE = type

定时（时钟）： 52

实例：

```
(a) INT     3   ;一字节指令：11001100

(b) INT     2   ;两字节：  11001101 00000010
    INT     67  ;两字节：  11001101 01000011
    IMM__44 EQU 44
    INT     IMM__44 ; 两字节：  11001101 00101100
```

注意： 操作数必须是立即数，不能是寄存器或存储器操作数。

标志： 影响　　IF, TF

说明： INT 将标志寄存器保护进栈（与 PUSHF 相似），清除 TF 和 IF 标志，然后用一个间接调用将控制转给256个矢量元素中的任一个。这条指令的 1 字节格式产生一个类型 3 的中断。

INTO
(Interrupt if overflow)——溢出中断指令

操作： 如果溢出标志 OF 为 0，不产生任何操作；若 OF 标志为 1，则 SP 减 2，保护所有的标志进栈，将陷阱标志 TF 和中断标志 IF 复位，SP 再减 2，把 CS 的当前指令保护进栈，然后把类型 4 中断的双字中断矢量的第二个字（段基地址）填入 CS 寄存器，SP 再减 2，并将 IP 指令指示器的当前内容（指面 INTO 下条指令）保存进栈，然后将类型 4 双字中断矢量的第一字填入 IP。类型 4 双字中断矢量位于绝对地址单元 16(10 H）处。类型 4 双字中断矢量的第一个字是处理类型 4 中断服务“过程”（程序）的偏移地址。这时段基地址已在 CS 中，于是完成一次溢出处理“过程”（程序）的“长调用。

参看 INT, IRET, PUSHF.

```
if (OF) = 1 then
  (SP) ← (SP) – 2
  ((SP) + 1:(SP)) ← FLAGS
  (IF) ← 0
  (TF) ← 0
  (SP) ← (SP) – 2
  ((SP)) + 1:(SP)) ← (CS)
  (CS) ← (12H)
  (SP) ← (SP) – 2
  ((SP) + 1:(SP)) ← (IP)
  (IP) ← (10H)
```

编码：

1 1 0 0 1 1 1 0

定时（时钟）： 52

实例： INTO

标志： 不影响

说明： 如果OF标志置1，则INTO指令把标志寄存器保存进栈，清除TF和IF标志，然后用一次间接调用，通过矢量中断类型4（在1OH单元处）把控制转给溢出中断处理过程；如果OF标志是0，则不进行操作，继续执行下一条指令。

IRET
(Interrupt return) ——中断返回指令

操作： 用存放在栈顶相邻两单元中的字填入指令指示器IP，然后SP加2，并且新栈顶相邻两单元中的字填入CS寄存器，这样就使控制转回到被中断的点上去；SP再加2，将栈顶单元中存放的标志取回标志寄存器（参看POPF.指令），SP再次加2。

```
(IP) ← ((SP) + 1:(SP))
(SP) ← (SP) + 2
(CS) ← ((SP) + 1:(SP))
(SP) ← (SP) + 2
FLAGS ← ((SP) + 1:(SP))
(SP) ← (SP) + 2
```

编码：

1 1 0 0 1 1 1 1

定时（时钟）： 24

实例： IRET

标志： 影响所有标志

说明： IRET 指令使控制返回到前面的中断操作保存栈的返回地址，并恢复保存在栈中的标志寄存器的内容。

JA and JNBE
(Jump if not below nor equal, or jump if above)
——“高于”和“不低于/“不等于”转移指令

操作： 若进位标志CF和全零标志ZF均为0，则将从这条指令末尾到目标标号之间的距离加到指令器IP中去，实现转移。若(CF) = 1 或(ZF) = 1，则不转移而顺序执行下一条指令。

IF (CF)|(ZF) = 0 then
(IP) ← (IP) + disp（符号扩展到16位）

编码：

0 1 1 1 0 1 1 0	disp

定时（时钟）： 产生转移 8
不产生转移 4

实例：

```
JA   TARGET_LABEL
JNBE  TARGET_LABEL
```

标志： 不影响

说明： 当“高于”或“不低于/等于”时，JA或JNBE 指令将控制转移给目标操作数。

注意： 目标标号必须在距离该指令的－128到＋127字节范围内。“高于”和低于”指的是两个无符号数之间的关系，“大于”和“小于”指的是两个有符号数之间的关系。

JAE and JNB
(Jump if not below, or jump if above or equal)
——“高于/等于”和“不低于”转移指令

操作： 若进位标志CF为0，则将该指令末尾到目标标号的距离加到指令指示器IP中去，并实现转移。若（CF）= 1，不产生转移，继续执行下一条指令。

```
if (CF) = 0 then
(IP) ← (IP) + disp（符号扩展到16位）
```

编码：

01110011	disp

定时（时钟）： 产生转移 8
不产生转移 4

实例：

```
JNB  TARGET_LABEL
JAE  TARGET_LABEL
```

标志： 不影响

说明： 当“高于/等于”或“不低于”时，JAE或者JNB指令将控制转移给目标操作数。

注意： JAE和JNB的目标标号必须在距离该指令的－128到＋127字节范围内。“高于”和“低于”指的是两个无符号数之间的关系。“大于”和“小于”指的是两个有符号数之间的关系。

JB and JNAE
(Jump if below, or jump if not above nor equal)
——“低于” 和“不高于”/“不等于”转移指令

JC
(Jump if carry)——有进位转移指令

操作： 若进位标志置1，则将从这条指令末尾到目标标号之间的距离加到指令指示器IP中去，实现转移。若(CF) = 0，不发生转移，顺序执行下一条指令。

```
if (CF) = 1 then
  (IP) ← (IP) + disp    (符号扩展到16位)
```

编码：

01110010	disp

定时(时钟)：发生转移　　8
　　不发生转移　　4

实例：

```
JB   TARGET_LABEL
JNAE  TARGET_LABEL
JC   TARGET_LABEL
```

标志：不影响

说明：当“低于”或“不高于/不等于”时，JB（或JNAE)将控制转移给目标操作数。

注释：目标标号必须在距离此指令的－128到＋127字节范围内,“在上”和“在下”指的是两个无符号数之间的关系,“大于”和“小于”指的是两个有符号数之间的关系。

JBE and JNA
(Jump if below or equal, or jump if not above)
“低于 等于”和“不高于”转移指令

操作：如果进位标志或0标志之中的任一个被置位，则把这条指令末尾到目标标号的距离加到指令指示器IP中去，以实现转移。若两个标志均为0，则(CF) = 0和(ZF) = 0，则不产生转移。

if (CF)|(ZF) = 1 then
(IP) ← (IP) + disp（符号扩展到16位）

编码：

01110110	disp

定时(时钟)： 发生转移 8
不发生转移 4

实例：

```
JBE  TARGET_LABEL
JNA  TARGET_LABEL
```

标志： 不影响

说明： 当“低于/等于”或“不高于”时，JBE（或 JNA）将控制转移给目标操作数。

注释： 目标标号必须在距离此条指令的－128到＋127字节的范围内。“在上”和“在下”指的是两个无符号数之间的关系，“大于”和“小于”指的是两个有符号数之间的关系。

JCXZ

(Jump if CX is zero) —— “CX＝0”转移指令

操作： 若计数寄存器（CX）为0，则把这条指令的末尾到目标标号的距离加到指令指示器中去，以实现转移。

```
if (CX) = 0 then
  (IP) ← (IP) + disp (符号扩展到16位)
```

编码：

1 1 1 0 0 0 1 1	disp

定时(时钟)：发生转移　　9
　　不发生转移　　5

实例：　JCXZ　TARGET_LABEL

标志：不影响

说明：若CX寄存器的内容为0，JCXZ指令将控制转给目标操作数。

注意：目标标号必须在距离这条指令的－128到＋127字节的范围内。

JE and JZ
(Jump if equal, jump if zero) "等于"和"全0"转移指令

操作：如果最后一次操作的结果为0，那么ZF标志就被置成1，当(ZF)＝1时，就把这个指令末尾到目标标号之间的距离加到指令指示器IP中去，以实现转移。
若(ZF)＝0，不发生转移

if (ZF) = 1 then
(IP) ← (IP) + disp (符号扩展到16位)

编码：

01110100	disp

定时(时钟)：发生转移　　8
　　不发生转移　　4

实例：

```
1)    CMP   CX,   DX
      JE    LAB2
      INC   CX
LAB2:
;只有当CX ≠ DX时，CX 才加1

2)    SUB   AX,   BX
      JZ    EXACT
      ;只有结果为0，即AX = BX时，才发生转移
      .
      .
      .
EXACT:
```

标志： 不影响

说明：当最后一次操作的结果为0时，这条指令将控制转给目标操作数。

注释：目标标号必须在距离这条指令的－128到＋127字节范围内。“在上”和“在下”指的是两个无符号数之间的关系，“大于”和“小于”指的是两个有符号数之间的关系。

JG and JNLE

(Jump if not less nor equal, or jump if greater)

——“大于”和“不小于/不等于”转移指令

操作： 如果零标志复位且符号标志与溢出相等（即二者都为0或为1），则把从这条指令的末尾到目标标号的距离加到指令指示器IP中去，以实现转移。若 (ZF) = 1 或 (SF) ≠ (OF)，则不产生转移。

if ((SF)⊕(OF))∥(ZF) = 0 then
(IP) ← (IP) + disp (符号扩展到16位)

编码：

01111111	disp

定时(时钟)： 发生转移 8
不发生转移 4

实例：

```
JG   TARGET_LABEL
JNLE  TARGET_LABEL
```

标志： 不影响

说明： 当“大于”或“不小于/不等于”时，JG或JNLE指令将控制转移给目标操作数。

注释： 目标标号必须在距离这条指令的－128到＋127字节范围内。“在上”和“在下”指的是两个无符号数之间的关系，“大于”和“小于”指的是两个有符号数之间的关系。

JGE and JNL
(Jump if not less, or jump if greater or equal)
—“大于/等于”和“不小于”转移指令

操作： 如果符号标志等于溢出标志，则把这条指令末尾到目标标号之间的距离加到指令指示器IP中去，以实现转移。若(SF) ≒ (OF)，不产生转移。

if (SF)‖(OF) = 0 then
(IP) ← (IP) + disp (符号扩展到16位)

编码：

01111101	disp

定时(时钟)： 发生转移 8
不发生转移 4

实例：

```
JGE  TARGET_LABEL
JNL  TARGET_LABEL
```

标志： 不影响

说明： 当“大于/等于”或“不小于”时，JGE或JNL指令控制转移给目标操作数。

注释： 目标标号必须在距离此条指令的－128到＋127字节范围内。“在上”和“在下”指的有两个无符号数之间的关系，“大于”和“小于”指的是两个有符号数之间的关系。

JL and JNGE
(Jump on less, or jump on not greater nor equal)
—“小于”和“不大于/不等于”转移指令

操作： 只有当符号标志不等于溢出标志——即 (SF) ≒ (OF) 时，才发生转移。也可以说是 (SF) 异或 (OF) = 1，若(SF)≒(OF)

则此条指令末尾到目标标号之间的距离加到指令指示器IP中去。若(SF) = (OF)，不产生转移。

```
if (SF)‖(OF) = 1 then
  (IP) ← (IP) + disp (符号扩展到16位)
```

编码：

01111100	disp

定时(时钟)： 发生转移 8
不发生转移 4

实例：

```
JL   TARGET_LABEL
JNGE  LABEL_TARGET
```

标志： 不影响

说明： 当“小于”或“不大于/不等于”时，将控制转移给目标操作数。

注释： 目标标号在距离此指令的－128到＋127字节范围内。“在上”和“在下”指的是两个无符号数之间的关系。“大于”和“小于”系指两个有符号数之间的关系。

JLE and JNG

(Jump if less or equal, or jump if not greater)

——“小于/等于”和“不大于”转移指令

操作：如果零标志被置位或符号标志不等于溢出标志，则把从这条指令末尾到目标标号之间的距离加到指令指示器IP中去，以实现转移，若(ZF)=0，和(SF)=(OF)，则不产生转移。

```
if ((SF)‖(OF))‖(ZF)=1 then
  (IP)←(IP) + disp (符号扩展到16位)
```

编码：

01111110	disp

定时(时钟)：产生转移 8
不产生转移 4

实例：

```
JLE  TARGET_LABEL
JNG  TARGET_LABEL
```

标志：不影响

说明：当“小于/等于”或“不大于”时，JLE (或JNG)将控制转给目标操作数。“在上”和“在下”指的是两个无符号数之间的关系，“大于”和“小于”指的是两个有符号数之间的关系。

JMP

(Jump) ——无条件转移指令

操作：在所有的段间转移和段内（或群内）间接转移中，用目标的偏移地址去置换指令指示器IP中的内容。

当转移为一个直接的段内或群内转移时，把这条指令末尾到目标标号的距离加到指令指示器IP中去。

直接段间转移首先用跟在指令操作码后的第二个字去置换CS的内容，然后再用跟在指令操作码后的第一个字去替换IP中的内容。

间接段间转移首先用跟在指令中指示数据地址字节后的第二个字送入CS，然后把跟在指令中指示数据地址字节后的第一个字送入IP。

编码：有以下五种格式

段内或群内直接转移：

11101001	disp-low	disp-high

DEST = (IP) + disp

定时(时钟)：7

实例：

段内直接短转移：

11101011	disp

DEST = (IP) + disp(符号扩展到16位)

定时(时钟)：1

实例：

```
JMP  TARGET_LABEL
JMP  SHORT  NEAR_LABEL
```

注释：目标标号必须在距离此指令的－128到＋127字节范围内。

段间直接转移：

1 1 1 0 1 0 1 0	offset-low	offset-high	seg-low	seg-high

DEST = offset, SEG = seg

定时(时钟)：7

实例：

```
JMP   LABEL_DECLARED_FAR
JMP   FAR  PTR  LABEL_NAME
JMP   FAR  PTR  NEAR_LABEL
```

段内间接转移：

1 1 1 1 1 1 1 1	mod 1 0 1 r/m

DEST = (EA), SEG = (EA + 2)

定时(时钟)：16＋EA

实例：

```
JMP   VAR_DOUBLEWORD
JMP   DWORD  PTR  [BX] [SI]
JMP   ALPHA [BP] [DI]
```

段内或群内间接转移：

1 1 1 1 1 1 1 1	mod 1 0 0 r/m

DEST = (EA)

定时(时钟)：7

实例：

```
JMP   TABLE [BX]
JMP   WORD PTR  [BX] [DI]
JMP   BETA_WORD
JMP   AX
JMP   SI
JMP   BP
;      这些指令用命名的寄存器内容去置换指令指示器的内
;容。这将引起直接到 CS 段位移量指定字节的转移。这种转
;移与直接段内转移是有区别的，直接段内转移是自相对的，
;将偏移地址加到IP中去。
```

说明：JMP 指令无条件地将控制转移给目标操作数。这种转移总是相对于 CS 寄存器中段基地址的转移。直接转移指令是直接地使用跟在指令操作码后的位移量字（若为长转移，还有段字），间接转移指令使用跟在指令操作码字节后的字节指定地址单元中的内容。

JNA and JBE

(Jump if below or equal, or jump if not above)

——“不高于”和“低于/等于”转移指令

操作： 如果进位标志或零标志二者之一为 1 时，则这条指令末尾开始到目标标号的距离加到指令指示器中去，以实现转移。如果 (CF) 与 (ZF) 均为 0，则不产生转移。

```
if (CF)|(ZF) = 1 then
  (IP) ← (IP) + disp   (符号扩展到16位)
```

注释：目标标号必须在距离此指令的－128到＋127字节范围内。

段间直接转移：

1 1 1 0 1 0 1 0	offset-low	offset-high	seg-low	seg-high

DEST = offset, SEG = seg

定时(时钟)：7

实例：

```
JMP   LABEL_DECLARED_FAR
JMP   FAR  PTR  LABEL_NAME
JMP   FAR  PTR  NEAR_LABEL
```

段内间接转移：

1 1 1 1 1 1 1 1	mod 1 0 1 r/m

DEST = (EA), SEG = (EA + 2)

定时(时钟)：16＋EA

实例：

```
JMP   VAR_DOUBLEWORD
JMP   DWORD  PTR  [BX][SI]
JMP   ALPHA[BP][DI]
```

段内或群内间接转移：

1 1 1 1 1 1 1 1	mod 1 0 0 r/m

DEST = (EA)

定时(时钟)：7

编码：

01110010	disp

定时(时钟)： 发生转移 8
不发生转移 4

实例：

```
JB   TARGET_LABEL
JNAE  TARGET_LABEL
```

标志： 不影响

说明： 当“不高于/不等于”或“低于”时，JNAE（或JB)指令将控制转移给目标操作数。

注释： 目标标号必须在距离此指令的－128到＋127字节范围内。“在上”和“在下”指的是无符号数之间的相对关系，“大于”和“小于”指的是有符号之间的相对关系。

JNB and JAE
(Jump if not below, or jump if above or equal)
—— “不低于”和“高于/等于”转移指令

JNC
(Jump if no carry) —— 进位标志为“0”转移指令

操作： 若进位标志为0，则把这条指令末尾到目标标号之间的距离加到指令指示器中去，以实现转移。若(CF) =1，不发生转移。

if (CF) = 0 then
(IP) ← (IP) + disp（符号扩展到16位）

编码：

01110011	disp

定时(时钟)： 发生转移 8
不发生转移 4

实例：

```
JNB   TARGET_LABEL
JAE   TARGET_LABEL
JNC   TARGET_LABEL
```

标志： 不影响

说明： 当“不低于”或“高于/等于”时，JNB（或JAE）将控制转给目标操作数。

注释： 目标标号必须在距离此条指令为中心的－128到＋127字节范围内。“在上”和“在下”指的是两个无符号数之间的关系，“大于”和“小于”指的是两个有符号数之间的关系。

JNBE
(Jump if not below nor equal)
——“不低于不等于”转移指令

操作： 若进位标志或零标志均没有被置位，则把这条指令末尾到目

标标号之间的距离加到指令指示器IP中去，以实现转移。

若 (CF) = 1或 (ZF) = 1，不产生转移。

if (CF)|(ZF) = 0 then
(IP) ← (IP) + disp (符号扩展到16位)

编码：

01110111	disp

定时(时钟)：

发生转移	8
不发生转移	4

实例：

```
JNBE  TARGET_LABEL
JA    TARGET_LABEL
```

标志： 不影响

说明： 当“不低于不等于”时，JNBE或JA指令将控制转移给目标操作数。

注释： 目标标号必须在距离此条指令的－128到＋127字节范围内。“在上”和“在下”指的是两个无符号数之间的相对关系，“大于”和“小于”指的是两个有符号数之间的相对关系。

JNE and JNZ
(Jump if not equal, or jump if not zero)
——“不等于”和“不为零”转移指令

操作： 如果零标志复位，则把这条指令末尾到目标标号的距离加到指令指示器IP中去。若 (ZF) = 1不发生转移。

if (ZF) = 0 then
(IP) ← (IP) + disp (符号扩展到16位)

编码：

01110101	disp

定时(时钟)： 发生转移 8
不发生转移 4

实例：

```
JNE  TARGET_LABEL
JNZ  TARGET_LABEL
```

标志： 不影响

说明： 当“不等于/不为零”时，JNE（或JNZ）指令将控制转移给目标操作数。

注释： 目标标号必须在距离此条指令的—128到+127字节范围内。“在上”和“在下”指的是两个无符号数之间的相对关系，“大于”和“小于”指的是两个有符号数之间的相对关系。

JNG and JLE

(Jump if not greater, or jump if less or equal)

——“不大于”和小于/等于”转移指令

操作： 若零标志置位，或符号标志不等于溢出标志，则把这条指令

末尾到目标标号的距离加到指令指示器 IP 中去，以实现转移。若 (ZF) = 0及 (SF) = (OF)，则不发生转移。

if ((SF)⊕(OF))∨(ZF) = 1 then
(IP) ← (IP) + disp（符号扩展到16位）

编码：

01111110	disp

定时(时钟)： 发生转移 8
不发生转移 4

实例：

```
JLE   TARGET_LABEL
JNG   TARGET_LABEL
```

标志： 不影响

说明： 当“不大于”或“小于/等于”时，JNG（或JLE）指令将控制转移给目标操作数。

注释： 目标标号必须在距离此指令的－128到＋127字节范围内。“在上”和“在下”系指两个无符号数之间的关系，“大于”和“小于”系指两个有符号数之间的关系。

JNGE and JL

(Jump if less, or jump if not greater nor equal)

—— “不大于/不等于”和“小于”转移指令

操作：如果符号标志SF不等于溢出标志OF，则把这条指令末尾到目标标号的距离加到指令指示器IP中去，以实现转移。若(SF) = (OF)，不产生转移。

if (SF) ‖ (OF) = 1 then
(IP) ← (IP) + disp(符号扩展到16位)

编码：

01111100	disp

定时(时钟)：发生转移 8
不发生转移 4

实例：

```
JL   TARGET_LABEL
JNGE   TARGET_LABEL
```

标志：不影响

说明：当“不大于/不等于“或”小于”时，JNGE（或JL)指令将控制转移到目标操作数。

注释：目标标号的范围必须在距离此条指令的－128到＋127字节范围内。“在上”和“在下”指的是两个无符号数之间的关系，“大于”和“小于”指的是两个有符号数之间的关系。

JNL and JGE

(Jump if not less, or jump if greater or equal)

——“不小于”和“大于/等于”转移指令

操作：若符号标志SF等于溢出标志OF，则把此条指令末尾到目标标号的距离加到指令指示器中去，以实现转移。若 (SF) ≠ (OF)，不产生转移。

if (SF)‖(OF) = 0 then
(IP) ← (IP) + disp （符号扩展到16位）

编码：

01111101	disp

定时(时钟)：发生转移 8
不发生转移 4

实例：

```
JGE   TARGET_LABEL
JNL   TARGET_LABEL
```

标志：不影响

说明：当“不小于”或“大于/等于”时，JNL（或JGE)指令将控制转移给目标操作数。

注释：目标标号必须在距离此条指令的－128到＋127字节范围内。“在上”和“在下”指的是两个无符号数之间的关系，“大于”和“小于”是指两个有符号数之间的关系。

JNLE and JG

(Jump if not less nor equal, or jump if greater)

—— “不小于/不等于“和“大于”转移指令

操作：若零标志ZF复位及符号标志SF等于溢出标志OF（即二者均为0或均为1），则把这条指令末尾到目标标号之间的距离加到指令指示器IP中去。若(ZF) = 1或(SF) ≠ (OF)，则不发生转移。

```
if ((SF)∥(OF))∥(ZF) = 0 then
  (IP) ← (IP) + disp（符号扩展到16位）
```

编码：

01111111	disp

定时(时钟)：发生转移 8
不发生转移 4

实例：

```
JG   TARGET_LABEL
JNLE  TARGET_LABEL
```

标志：不影响

说明：当“不小于/不等于”或“大于”时，JNLE（或JG）指令将控制转移给目标操作数。

注释：目标标号必须在距离此条指令的－128到＋127字节范围内。“在上”和“在下”指的是两个无符号数之间的关系，“大于”和“小于”指的是两个有符号数之间的关系。

JNO

(Jump on not overflow) ——无溢出转移指令

操作： 若溢出标志OF为1，不产生转移；若(OF) = 0，则把这条指令末尾到目标标号的距离加到指令指示器IP中去，以实现转移。

if (OF) = 0 then
(IP) ← (IP) + disp（符号扩展到16位）

编码：

01110001	disp

定时(时钟)： 发生转移 8
不发生转移 4

实例： JNO TARGET_LABEL

标志： 不影响

说明： 当运算未产生溢出时，JNO指令把控制转移给目标操作数。

注释： 目标标号必须在距离这条指令的－128到＋127字节范围内。

JNP and JPO
(Jump on no parity or jump if parity odd)
——"无奇偶性和奇偶性为奇"转移指令

操作： 若奇偶性标志PF置1，即最后影响PF标志的指令将奇偶性标志置为偶性时，则不发生转移。若(PF) = 0，则把这条指令末尾到目标标号的距离加到指令指示器IP中去，以实现转移。

if (PF) = 0 then
(IP) ← (IP) + disp（符号扩展到16位）

编码：

01101011	disp

定时(时钟) 发生转移 8
不发生转移 4

实例：

1) JNP TARGET_LABEL
2) JPO TARGET_LABEL

标志： 不影响

说明： 当无奇偶性（或奇偶性为奇）时，JNP（或JPO）将控制转给目标操作数。

注释： 目标标号必须在距离此指令的－128到＋127字节范围内。

JNS
(Jump on not sign, jump if positive)
——无符号（符号标志为零）转移指令

操作： 若符号标志SF没有置1，则把这条指令末尾到目标标号的距离加到指令指示器IP中去，以实现转移。若 (SF) = 1，则不发生转移。

if (SF) = 0 then
(IP) ← (IP) + disp（符号扩展到16位）

编码：

01111001	disp

定时(时钟)： 发生转移 8
不发生转移 4

实例： JNS TARGET_LABEL

标志： 不影响

说明： 当(SF) = 0时，JNS指令将控制转移给目标操作数。

注释： 目标标号必须在距离此指令的－128到＋127字节范围内。

JO

(Jump on overflow) ——溢出转移指令

操作： 若溢出标志OF为1，则把这条指令末尾到目标标号之间的距离加到指令指示器IP中去，以实现转移。若 (OF) = 0，则不发生转移。

if (OF) = 1 then
(IP) ← (IP) + disp (符号扩展到16位)

编码：

01110000	disp

定时(时钟)： 发生转移 8
不发生转移 4

实例： JO TARGET_LABEL

标志： 不影响

说明： 当溢出标志OF置位时，JO指令把控制转移给目标操作数。

注释： 目标标号必须在距离此指令的－128到＋127字节范围内。

JP and JPE

(Jump on parity, or jump if parity even)

——"有奇偶性和奇偶性为偶"转移指令

操作： 当奇偶性标志PF置1，则把这条指令末尾到目标标号之间的距离加到指令指示器IP中去，以实现转移。若(PF)＝0，不产生转移。

if (PF) = 1 then
(IP) ← (IP) + disp（符号扩展到16位）

编码：

01110010	disp

定时(时钟)： 发生转移 8
不发生转移 4

实例：

1) JP TARGET_LABEL
2) JPE TARGET_LABEL

标志：不影响

说明：当奇偶性标志置位，或奇偶性为偶时，JP或JPE指令将控制转移给目标操作数。

注释：目标标号必须在距离此指令的－128到＋127字节范围内。

JPE and JP
(Jump on parity, or jump if parity even)

——"奇偶性为偶和有奇偶性"转移指令

操作：若奇偶性标志PF置1，把这条指令末尾到目标标号之间的距离加到指令指示器IP中去，以产生转移。若 (PF) = 0，不产生转移。

```
if (PF) = 1 then
  (IP) ← (IP) + disp（符号扩展到16位）
```

编码：

01111010	disp

定时(时钟)：发生转移 8
不发生转移 4

实例：

```
1) JP   TARGET_LABEL
2) JPE  TARGET_LABEL
```

标志：不影响

说明：当“奇偶性为偶”或“有奇偶性”，即(PF)=1时，JPE或JP指令将控制转移给目标操作数。

注释：目标标号必须在距离此指令的－128到＋127字节范围内。

JPO and JNP
(Jump on no parity or jump if parity odd)
——奇偶性为奇和无奇偶性”转移指令

操作：若奇偶性标志PF置1，则不产生转移。反之，若 (PF) = 0，则把这条转移指令末尾到目标标号之间的距离加到指令指示器IP中去，以实现转移。

```
if (PF) = 0 then
  (IP) ← (IP) + disp (符号扩展到16位)
```

编码：

01111011	disp

定时(时钟)：发生转移 8
不发生转移 4

实例：

```
1) JNP   TARGET_LABEL
2) JPO   TARGET_LABEL
```

标志：不影响

说明：当“奇偶性为奇”或“无奇偶性”时，JPO或JNP指令将控制转移给目标操作数。

注释：目标标号必须在距离此条指令的－128到＋127字节范围内。

JS
(Jump on sign) ——符号标志置1转移指令

操作：若符号标志SF为1，则把这条指令末尾到目标标号之间的距离加到指令指示器IP中去，以实现转移。

```
if (SF) = 1 then
  (IP) ← (IP) + disp(符号扩展到16位)
```

编码：

01111000	disp

定时(时钟)：产生转移　　8
不产生转移　　4

实例：　JS　TARGET_LABEL

标志：不影响

说明：当“符号标志置1”时，JS指令将控制转移给目标操作数。

注释：目标标号必须在距离此指令的－128到＋127字节范围内。

JZ and JE
(Jump if equal, jump if zero) ——为零和相等转移指令

操作：若零标志ZF置位，则把这条指令末尾到目标标号的距离加到指令指示器IP中去，以实现转移。

```
if (ZF) = 1 then
  (IP) ← (IP) + disp （符号扩展到16位）
```

编码：

01110100	disp

定时(时钟)：产生转移 8

不产生转移 4

实例：

1)
```
CMP  CX,  DX
JE   LAB2
INC  CX
LAB2:
; 只有当CX = DX 时，CX 才加 1
```

2)
```
SUB  AX,  BX
JZ   EXACT
;若结果为 0，即AX = BX，则发生转移
.
.
.
EXACT:
```

标志：不影响

说明：当结果为0（或两数相等）时，JZ（或JE)指令将控制转移给目标操作数。

注释：目标标号必须在距离此条指令的—128到+127字节范围内。

LAHF

(Load AH from flags) ——取标志到AH寄存器指令

操作： 将标志传送到AH寄存器中的指定位中去，符号标志SF的状态送入AH的第7位，零标志SF填入AH的第6位，辅助进位标志AF填入AH的第4位，奇偶性标志PF填入AH的第2位，进位标志CF填入AH的第0位；AH的第1、3、5位是不确定的，有时可能为1，有时可能为0。

(AH) ← (SF):(ZF):X:(AF):X:(PF):X:(CF)

编码：

10011111

定时(时钟)： 4

实例： LAHF

标志： 不影响

说明： LAHF指令将标志寄存器中的SF, ZF, AF, PF和CF标志位传送到AH寄存器中的指定位（7、6、4、2、0位）中去。在把8080代码翻译成为8086代码时，用这条指令来形成8086的相应标志位，X表示不确定的位。

LDS

(Load data segment register) ——取指示器到DS指令

操作： 1）用双倍字长存储器操作数的低位地址中的字去置换指定寄存器的内容，即

(REG) ← (EA)

2）用双倍字长存储器操作数的高位地址中的字 去 置换 DS 寄存器的内容，即

(DS) ← (EA + 2)

编码：

1 1 0 0 0 1 0 1	mod reg r/m

当mod≠11，此指令是有效的（若mod=11,即寄存器方式，则操作不确定）

定时(时钟)： 16 + EA

实例：

```
LDS   BX,  ADDR_TABLE [SI]
LDS   SI,  NEWSEG [BX]
```

标志： 不影响

说明： LDS 指令用来将源操作数中的“指示器—目标”（它是一个包含有偏移地址及段地址的32位的目标）传送到“一对寄存器”中去。“指示器—目标”中的段地址传 送 到 DS寄存器中去。“指示器—目标”中的偏移地址可传送到任何16位的通用寄存器、指示器或变地址寄存器中去。

LEA
(Load effective address) ——取有效地址指令

操作：用给出的变量或标号或表达式的偏移地址(即有效地址EA)去取代指定寄存器中原有的内容。

(REG) ← EA

编码：

10001101	mod reg r/m

当mod≠11，存储器操作数有效（若mod = 11，此操作不确定）

定时(时钟)：2 + EA

实例：

```
LEA   BX,   VARIABLE_7
LEA   DX,   BETA [BX] [SI]
LEA   AX,   [BP] [DI]
```

标志：不影响

说明：LEA指令将源操作数的偏移地址EA传送到目的操作数中去。这里源操作数必须是一个存储器操作数，而目的操作数必须是任一个16位的通用寄存器，指示器或变址寄存器。

LEA指令允许源操作数带下标变量，这对于带 OFFSET 操作符的MOV指令是不允许的，而且在随后的操作中，使用已经在段中定义的变量偏移地址是不允许改变的。然而，如果利用最后设置的 ASSUME 伪指令，群是唯一的可能访问的途径，则LEA指令将把群的偏移地址计算进去。

LES

(Load extra-segment register)

——取指示器到附加段寄存器（ES）指令

操作：

1）用双字存储器操作数的低位字去置换指定寄存器的内容

(REG) ← (EA)

2）用双字存储器操作数的高位字去置换 ES 寄存器的内容

(ES) ← (EA + 2)

编码：

1 1 0 0 0 1 0 0	mod reg r/m

当mod≠11，存储器操作数是有效的（若 mod = 11，寄存器操作数不确定）

定时(时钟)： 16 + EA

实例：

```
LES   BX,   ADDR_TABLE [SI]
LES   DI,   NEWSEG [BX]
```

标志： 不影响

说明： LES 指令将源操作数（必须是存储器操作数）中的“指示器—目标”（一个包含有偏移地址及段地址的32位的目标）传送到“一对目的寄存器”中去。这里段地址被送到ES附加段寄存器中去，偏移地址可送到16位通用寄存器指示器、变址寄存器中的任何一个中去。

LOCK ——总线锁定前缀

操作：无

编码：

11110000

定时(时钟)：2

实例：LOCK

标志：不影响

说明：一字节的LOCK前缀可以放在任何一条指令的前面，它使处理器在指令执行期间保持“总线锁定”信号$\overline{\text{LOCK}}$。在多处理机系统中，需要用这个前缀来对共享资源进行强迫控制。通常这种方法由操作系统软件来实现，但操作系统要求一定的硬件支援。用“锁定交换”（通称为“检查锁定和置位锁定”）的原理就足可以完成对共享资源的控制了。

该指令对完成寄存器与存储器交换方面的任务最为有效。用下列的程序段可以实现一个简单的软件锁定任务。

```
Check:  MOV   AL,1          ; 置AL为1（意味着要锁定）
  LOCK  XCHG  Sema,AL       ; 检查和置位锁定
        TEST  AL,AL         ; 根据AL置标志位
        JNZ   Check         ; 若锁定已置位，重测
        .
        .
        MOV   Sema,0        ; 当完成时，清除锁定
```

LOCK 前缀可与段修改前缀与/或重复前缀 REP 组合起来使用。但要注意与REP联用会出现的某些问题（参看 REP）

LODS

(Load byte or word string) ——取字节串或字串指令

操作：将源字节（或字）取入AL（或AX）寄存器中去，若方向标志DF是复位的，则源变址寄存器加1（或加2，对字串来说是这样）；否则，当DF标志置1时，SI减1（或减2）。

(DEST) ← (SRC)
if (DF) = 0 then (SI) ← (SI) + DELTA
else (SI) ← (SI)−DELTA

编码：

1 0 1 0 1 1 0 w

1) if w = 0 then SRC = (SI), DEST = AL, DELTA = 1
2) else SRC = (SI) + 1:(SI), DEST = AX, DELTA = 2

定时(时钟)：12

实例：

```
1) CLD   ;清除方向标志DF，于是SI将增量
   MOV   SI,  OFFSET  BYTE_STRING
   LODS  BYTE_STRING      ;SI ← SI+1
         .
         .
         .

2) STD   ;置位DF，于是SI将被减量
   MOV   SI,  OFFSET  WORD_STRING
   LODS  WORD_STRING      ;SI ← SI-2
   ; DF =1意味着变量WORD_STRING是字符串中最后的或
   ; 最高地址单元中的字的名称，在LODS指令中命名的操作
   ; 数仅由汇编程序使用，汇编程序用它来验证操作数属性和
```

；使用段寄存器内容的可达性。实际上LODS只用SI去
；指定那些要将其内容装入累加器的单元，而不用在源指令
；中给出的名字。

标志：不影响

说明：LODS指令把SI寄存器寻址的串元素字节（或字）传送到寄存器AL（AX）中去，并用DELTA去修正SI寄存器，以指向串中的下一个元素。由于每次重复该指令时要将累加器AL（或AX）中的内容冲掉，只有最后一个元素能保留下来，所以该指令一般是不重复的。

LOOP
(Loop, or iterate instruction sequence until count complete) ——循环或迭代控制指令

操作：计数寄存器CX减1，若新的CX值不为零，则把这条指令末尾到目标标号之间的距离加到指令指示器IP中去，以实现转移。若CX等于0，则不产生转移。

```
(CX) ← (CX)−1
if (CX) ≠ 0 then
  (IP) ← (IP) + disp（符号扩展到16位）
```

编码：

11100010

定时(时钟)：发生转移 9
不发生转移 5

实例：下面的指令序列用来计算一个非0数组的“检查和”

```
(1)     MOV   CX,   LENGTH   ARRAY
        MOV   AX,   0
        MOV   SI,   AX

NEXT:   ADD   AX,   ARRAY [SI]
        ADD   SI,   TYPE ARRAY
        LOOP  NEXT
        MOV   CKS,  AX

(2)     MOV   AX,   0
        MOV   BX,   1
        MOV   CX,   N   ;项目数
        MOV   DI,   AX

FIB:    MOV   SI,   AX
        ADD   AX,   BX
        MOV   BX,   SI
        MOV   FIBONACCI [DI],   AX
        ADD   DI,   TYPE FIBONACCI

LL:     LOOP  FIB
```

；从 FIB 到 LL 的指令将被执行 N 次，并将该序列的第一组 N ；项目，即 1，1，2，3，5，8，13，21……存入 FIBONACCI 数组中去。

标志：不影响

说明：LOOP 指令将CX计数寄存器的内容减 1，若 CX 内容不为 0，将控制转移给目的操作数；否则，不发生转移，顺序执行跟在LOOP之后的一条指令。

目标标号必须在距离此条指令的－128到127字节范围内。

LOOPE and LOOPZ
(Loop on equal, or loop on zero)

——“相等”或“为零”循环指令

操作：计数寄存器（CX）减1。如果零标志ZF置位及（CX）还没有减到零，则把这条指令末尾到目标标号之间的距离加到指令指示器IP中去，以实现转移。

如果（ZF）= 0或（CX）= 0，则不发生转移。

```
(CX) ← (CX)-1
if (ZF) = 1 and (CX) ≠ 0 then
  (IP) ← (IP) + disp（符号扩展到16位）
```

编码：

11100001	disp

定时(时钟)：发生转移 11

不发生转移 5

实例：下列指令序列用于在一个字节数组中找出第一个非零的项目

```
        MOV  CX,  LENGTH  ARRAY
        MOV  SI,  -1

NEXT:   INC   SI
        CMP   ARRAY [SI],  0
        LOOPE  NEXT
        JNE   OKENTRY
        .                    ; 如果整个数组为零，则到此
        .
        .
OKENTRY:                     ; SI 指出哪个项目是非零的项目
```

标志：不影响

说明：LOOPE，又称为LOOPZ指令，它将CX计数寄存器的内容减1，若CX内容不为0及ZF标志置位，则将控制转移给目的操作数；否则顺序执行下一条指令。

LOOPNE and LOOPNZ
(Loop on not zero, or loop on not equal)
——“不相等”和“不为零”循环指令

操作： 计数寄存器（CX）减1。如果新的（CX）不为零且零标志ZF是复位的，则把此条指令末尾到目标标号之间的距离加到指令指示器IP中去，以实现转移。若（CX）= 0或者若（ZF）= 1，则不发生转移。

(CX) ← (CX)−1
if (ZF) = 0 and (CX) ≠ 0 then
  (IP) ← (IP) + disp（符号扩展到16位）

编码：

11100000	disp

定时(时钟)： 发生转移 11
不发生转移 5

实例： 下面的指令序列用来求两个二字节数组之和，每个数组的长度为N。如果同时遇到两个数组中的元素都为零，就停止求和。表达式SI-1将给出此“非零和”数组的长度。

```
        MOV   AX,  0
        MOV   SI-1
        MOV   CX,  N
NONZER: INC   SI
        MOV   AL,  ARRAY1 [SI]
        ADD   ,ARRAY2 [SI]
        MOV   SUM [SI],  AX
        LOOPNZ  NONZER
```

下面的指令序列用来从一个“链接表”中找出最后的一个元素，该元素通常用于存放下一个元素的地址，现在是内容为0的字。这个字距离每个表元素的起点有一定的字节数。LINK用来代表这个距离的字节绝对数。

```
        LINK   EQU   7
        MOV    AX,   OFFSET HEAD_OF_LIST
        MOV    CX,   1000  ; 查找的项目最多为1000项
NEXT:   MOV    BX,   AX
        MOV    AX, [BX] + LINK
        CMP    AX,   0
        LOOPNE  NEXT
```

标志：不影响

说明：LOOPNE又称为LOOPNZ指令，它将CX寄存器的内容减1，若CX内容不为0，而且ZF标志为0，则循环，即将控制转移给目的操作数；否则，不循环而顺序地执行下一条指令。LOOPNZ和LOOPNE是同一指令的两种符号。

目标标号必须在距离此指令的－128到＋127字节范围内。

MOV
(Move) ——传送指令

一共有七种不同类型的传送指令。每种类型又可有多种使用方法，其编码依赖于被传送数据的类型及该数据所在的位置。汇编程序将根据这两个因素来生成正确的指令目标代码。如果目的操作数为寄存器，则指令操作码中相应于d的那位应置“1”，否则为“0”。如果类型为字，则指令操作码中对应w的那位应为“1”，否则为“0”。

编码：有以下七种格式

从累加器到存储器的数据传送：

1 0 1 0 0 0 1 w	addr-low	addr-high

If w=0 then SRC=AL, DEST=addr else SRC=AX, DEST=addr+1: addr

定时(时钟)： 10 + EA

实例：

```
MOV  ALPHA_MEM,  AX
MOV  GAMMA_BYTE,  AL

MOV  CS:DATUM_BYTE,  AL
MOV  ES:ARRAY [BX] [SI],  AX
```

（CS：和ES：为前缀字节。产生目标代码时，它将放在这条MOV指令的前面）

从存储器到累加器的数据传送：

1 0 1 0 0 0 0 w	addr-low	addr-high

If w=0 then SRC=addr, DEST=AL else SRC=addr + 1: addr, DEST=AX

定时(时钟)： 8 + EA

实例：

```
MOV  AX,  BETA_MEM
MOV  AL,  GAMMA_BYTE

MOV  AX,  ES:ARRAY [BX] [SI]
MOV  AL,  SS:OTHER_BYTE
```

（ES：和SS：为前缀字节，产生目标代码时，它将放在这条MOV指令的前面）

从存储器/寄存器操作数到段寄存器的数据传送：

1 0 0 0 1 1 1 0	mod 0reg r/m

if reg ≠ 01 then SRC=EA, DEST=REG 否则操作数不确定。

定时： 寄存器到寄存器 2
存储器到寄存器 8 + EA

实例：

```
MOV  ES,  DX
MOV  DS,  AX
MOV  SS,  BX
MOV  ES,  SS:NEW_WORD [DI]
```

注意： 在这里把CS作为目的操作数是非法的。

从段寄存器到存储器/寄存器的数据传送：

1 0 0 0 1 1 0 0	mod 0reg r/m

SRC=REG, DEST=EA,(DEST) ← (SRC)

定时(时钟)： 存储器到寄存器 9 + EA
寄存器到寄存器 2

实例：

```
MOV  DX,  DS
MOV  BX,  ES
MOV  ARRAY [BX] [SI],  SS
MOV  BETA_MEM_WORD,  DS
MOV  GAMMA, CS;  在这里CS作为源操作数是合法的。
```

(a) 从寄存器到寄存器，
(b) 从存储器/寄存器操作数到寄存器
(c) 从寄存器到存储器/寄存器操作数

1 0 0 0 1 0 d w	mod reg r/m	addr-low*	addr-high*

if d=1 then SRC=EA, DEST=REG else SRC=REG, DEST=EA

注有*记号的字节在寄存器到寄存器的传送（当mod=11寄存器方式时）指令中被略去。如MOV　　CX,　DX

还有，当存储器地址表达式是寄存器间接寻址且没有变量指定的位移量时，也要略去带*记号的字节。如

```
MOV     [BX] [SI],   DX
MOV     AX, [BP] [DI]
```

定时(时钟)：　(a) 2
(b) 8 + EA
(c) 9 + EA

实例：

```
(a) MOV   AX,   BX
    MOV   CL,   DH
    MOV   CX,   DI

(b) MOV   AX,   MEM_VALUE
    MOV   DX,   ARRAY [SI]
    MOV   DI,   MEM [BX] [DI]

(c) MOV   ARRAY [DI],   DX
    MOV   MEM_VALUE,   AX
    MOV   [BX] [SI],   DI
```

传送立即数到寄存器

1 0 1 1 w reg	data	data-high*

SRC=data, DEST=REG

*只有当 w = 1 时才有这个字节

定时(时钟)：　4

实例：

```
MOV   AX,   77
MOV   BX,   VALUE_14_IMM
MOV   SI,   EQU_VAL_9
MOV   DI,   618
```

传送立即数到存储器/寄存器操作数：

1 1 0 0 0 1 1 w	mod 000 r/m	data	data-high*

SRC=data, DEST=EA

*只有当 w = 1 时才有这个字节

定时(时钟)： 10 + EA

实例：

```
MOV   ARRAY [BX] [SI],   DATA_4
MOV   MEM_BYTE,   IMM_BYTE_3
MOV   BYTE   PTR   [DI],   66
MOV   MEM_WORD,   1999
MOV   BX,   84
MOV   DS:MEM_WORD [BP],   3989
```

（上面指令中的前缀字节 DS：（即00111110）在传送指令操作码第一字节1100011 w的前面）

标志： 不影响

MOVS
(Move byte string or move word string)
——字节串或字串的传送指令

操作： 把源字符串的内容传送给目的字符串，源字符串的偏移地址在源变址寄存器 SI 中，目的字符串的偏移地址在目的变址

寄存器DI中，目的单元必需在附加段中。如果方向标志为0，则SI和DI同时增量；否则方向标志DF = 1，SI和DI同时减量。增量或减量对字节串是1，而对字串是2。

```
(DEST) ← (SRC)
if (DF) = 0 then
  (SI) ← (SI) + DELTA
  (DI) ← (DI) + DELTA
else
  (SI) ← (SI)−DELTA
  (DI) ← (DI)−DELTA
```

编码：

1 0 1 0 0 1 0 w

if w = 0 then SRC = (SI), DEST = (DI), DELTA = 1
else SRC = (SI) + 1:(SI), DEST = (DI) + 1:(DI), DELTA = 2

定时(时钟)： 17

实例：

```
        MOV  SI,  OFFSET  SOURCE
        MOV  DI,  OFFSET  DEST
        MOV  CX,  LENGTH  SOURCE
REP     MOVS  DEST,  SOURCE
; 上述指令序列可将整个的源字符串传送到附加段中的目的
; 单元中去。源字符串在当前段寄存器指定的段中，目的单
; 元在附加段中。(在串操作中ES 寄存器总是用于 DI 操作
; 数)，請参看 REP。在字符串操作中指定的操作数只是汇
; 编程序来使用的。汇编程序用它们来验证操作数属性和当
; 前寄存器內容的可达性。实际上 MOVS 指令是把 SI 指向
; 的字节传送到ES中DI 指向的字节单元中去，在这种传送
; 过程中，并不需要使用在 MOVS 源指令中给出的名字。
```

标志：不影响

说明：MOVS 指令将源字符串（由SI寻址的）中的字节/字传送到目的字符串单元（由DI寻址的且在附加段中的）中去，并用DELTA去修正SI 和DI 寄存器的内容，使它们指向下一个元素。重复操作时，可进行存储器中的数据块传送。

MOVSB/MOVSW是MOVS指令的另一种格式的助记符。

MUL
(Multiply accumulator by register-or-memory; unsigned) ——无符号数的乘法指令

操作：用指定的操作数乘以累加器中的数，（若为字节，为AL；若为字，为AX）。若结果（乘积）的高一半为0，则将进位标志CF和溢出标志OF复位；反之，若结果（乘积）的一半不为0，均将CF和OF置位。

```
(DEST) ← (LSRC) * (RSRC),    此处 * 为无符号乘法
  multiply
if (EXT) = 0 then (CF) ← 0
else (CF) ← 1;
(OF) ← (CF)
```

编码：

1 1 1 1 0 1 1 w	mod 1 0 0 r/m

(a) if w = 0 then LSRC = AL, RSRC = EA, DEST = AX, EXT = AH
(b) else LSRC = AX, RSRC = EA, DEST = DX:AX, EXT = DX

定时(时钟)：8位　　71 + EA
16位　　124 + EA

实例：

a) MOV AL, LSRC_BYTE
MUL RSRC_BYTE ；结果在AX中

b1) MOV AX, LSRC_WORD
MUL RSRC_WORD
；高一半的结果在DX中，低一半的结果中AX中

b2) 用一个字去乘一个字节
MOV AL, MUL_BYTE
CBW ；将AL中的字节变换为AX中的字
MUL RSRC_WORD

注意： 上面给出的任何一个存储器操作数可以是一个具有正确属性的变址地址表达式。若 ARRAY 是 BYTE 属性，则 LSRC_BYTE 可以是 ARRAY [SI]；若 TABLE 是 WORD 属性，则 RSRC_WORD 可以是 TABLE [BX] [DI]

标志： 影响 CF, OF.
不确定 AF, PF, SF, ZF

说明： MUL 指令将 AL（或AX）累加器中的无符号数与源操作数中的无符号数相乘，把双倍字长的乘积送回累加器及其扩展寄存器中去。对于两个 8 位无符号数的乘法，16位的乘积在 AL 和AH中。对于两个16位的无符号乘法，32位的乘积存入 AX和DX中。如果乘积的高一半（在扩展部分EXT中）不为 0，则将CF 和OF都置 1。

NEG

(Negate, or form 2's complement) ——求补码指令

操作： 从全 1（对于字节为0 FFH，对于字为0 FFFFH ）中减去指定的操作数，然后再加 1，最后将结果存回指定的操作 数 中去。

(EA) ← SRC−(EA)
(EA) ← (EA) + 1 影响标志

编码：

1 1 1 1 0 1 1 w	mod 0 1 1 r/m

if w = 0 then SRC = 0FFH
else SRC = 0FFFFH

定时(时钟)： 寄存器 3
存储器 16+EA

实例： 1）若AL中的数为13H（00010011），则指令 NEG AL 使AL的内容变为－13H或者0EDH（11101101）。

2）若MEM_BYTE 中包含有0AFH（10101111），则指令 NEG MEM BYTE 使 MEM_BYTE 中包含有－0AFH或51H（01010001）。

3）若 SI中包含有2FC3H，则指令 NEG SI 使 SI 的内容变为0D03DH。

标志： 影响 AF, CF, OF, PF, SF, ZF

说明： NEG 指令执行0减去目的操作数的运算，加1后并将结果送回该目的操作数，这样就形成了指定操作数的补码。

NOP
(No operation) ——空操作指令

操作： 无

编码：

1 0 0 1 0 0 0 0

定时(时钟)： 3

实例： NOP

标志： 不影响

说明： NOP 指令使处理器执行空操作。每条 NOP 指令用3个时钟，NOP指令执行完后，接着执行其后继的指令。

NOT

(Not, or form 1's complement)

——求反码或逻辑非指令

操作： 从0FFH（对字节来说）或从0 FFFFH（对字来说）中减去指定的操作数，并将结果存回指定的操作数中去。

(EA) ← SRC−(EA)

编码：

1 1 1 1 0 1 1 w	mod 0 1 0 r/m

if w = 0 then SRC = 0FFH
else SRC = 0FFFFH

定时(时钟)： 寄存器 3
存储器 16+EA

实例：

1）若AH的内容是13H（00010011），则指令 NOT AH 使得AH变成0ECH（11101100）

2）若MEM_BYTE 中的内容是0 AFH（10101111），则 NOT MEM_BYTE 指令使 MEM_BYTE 中变成50H（01010000）。

3）若 DX 中有 2FC 3 H，则 指 定 NOT DX 使 DX 中变 0D0C3H，請再参看 NEG。

标志：不影响

说明：NOT 指令建立操作数的反码，并将结果送回操作数。

OR
(Or, inclusive)——逻辑或指令

操作：OR指令对操作数LSRC（左边的目的操作数）和RSRC(右边的源操作数）进行“按位或”操作，并将结果送回目的操作数中去。如果两个操作数中对应的位有一个为 1 或全为 1，则结果位为 1；否则，结果位为 0。该指令使进位标志 CF和溢出标志OF复位。

(DEST) ← (LSRC)∨(RSRC)
(CF) ← 0
(OF) ← 0

编码：有三种格式

存储器/寄存器操作与寄存器操作数：

0 0 0 0 1 0 d w	mod reg r/m

if d = 1 then LSRC = REG, RSRC = EA, DEST = REG
else LSRC = EA, RSRC = REG, DEST = EA

定时(时钟)：(a) 寄存器到寄存器 3
(b) 存储器到寄存器 9+EA
(c) 寄存器到存储器 16+EA

实例：

```
(a) OR   AH,   BL      ；结果在AH中，BL不变
    OR   SI,   DX      ；结果在SI中，BX不变
    OR   CX,   DI      ；结果在CX中，DI不变

(b) OR   AX,   MEM_WORD
    OR   CL,   MEM_BYTE [SI]
    OR   SI,   ALPHA [BX] [SI]

(c) OR   BETA [BX] [DI],   AX
    OR   MEM_BYTE,   DH
    OR   GAMMA [DI],   BX
```

立即操作数与累加器：

0 0 0 0 1 1 0 w	data	data if w=1

(a) if w = 0 then LSRC = AL, RSRC = data, DEST = AL
(b) else LSRC = AX, RSRC = data, DEST = AX

定时(时钟)： 立即操作数到寄存器　　4

实例：

```
a) OR   AL,   11110110B
   OR   AL,   0F6H

b) OR   AX,   23F6H
   OR   AX,   75Q
   OR   ,23F6H
```

立即操作数与存储器/寄存器操作数：

1 0 0 0 0 0 0 w	mod 0 0 1 r/m	data	data if w=1

LSRC = EA, RSRC = data, DEST = EA

定时(时钟)：(a) 立即操作数到寄存器　　4
(b) 立即操作数到存储器　　17+EA

实例：

a) OR AH, 0F6H
OR CL, 37
OR DI, 23F5H

b) OR MEM_BYTE, 3DH
OR GAMMA [BX] [DI], 0FACEH
OR ALPHA [DI], VAL_EQUD_33H

标志：影响　CF, OF, PF, SF, ZF.
不确定　AF

说明：OR指令用来实现两个操作数的“按位逻辑或”操作，并将结果送回到两个操作数之一中去。

OUT
(Output byte and output word) — “输出字节和字”指令

操作：用累加器的内容去置换目的端口中的内容。

(DEST) ← (SRC)

编码：有两种格式

固定的端口：(直接寻址)

1 1 1 0 0 1 1 w	port

if w = 0 then SRC = AL, DEST = port

else SRC = AX, DEST = port + 1:port
(0 < port < 255)

定时(时钟)：10

实例:

```
OUT   BYTE_PORT_VAL,AL   ;从AL中输出一个字节
OUT   WORD_PORT_VAL,AX   ;从AX中输出一个字
OUT   44,AX              ;从AX中输出一个字到端
                         ;口44
```

可变的端口:(间接寻址)

1 1 1 0 1 1 1 w

if w = 0 then SRC = AL, DEST = (DX)
else SRC = AX, DEST = (DX) + 1:(DX)

定时(时钟) 8

实例:

```
OUT   DX,AL    ;从AL中输出一个字节到DX指定的可变端
               ;口中去
OUT   DX,AX    ;从AX中输出一个字节到AX指定的可变端
               ;口中去
```

标志：不影响

说明：OUT指令将AL(或AX)中字节(或字)传送到输出端口。端口用指令中的数据字节指定,可访问固定的端口0～255;或者用DX寄存器中的端口地址指定,可访问可变的端口达64K个。

POP

(Pop word off stack into destination)

—从栈中把字弹出到目的操作数的指令

操作：

1) 用存放在栈顶中的字去替换目的操作数的内容

(DEST) ← ((SP) + 1:(SP))

2) 栈指示器SP加2

(SP) ← (SP) + 2

编码： 有以下三种格式

寄存器操作数：

0 1 0 1 1 reg

DEST = REG

定时(时钟)： 8

实例：

```
POP   CX
汇编程序产生的目标指令码为：0 1 0 1 1 0 0 1
POP   DX
汇编程序产生的目标指令码为：0 1 0 1 1 0 1 0
```

段寄存器操作数：

0 0 0 reg 1 1 1

if reg ≠ 01 then DEST = REG
否则操作不确定

注意：POP CS指令是非法的。

定时(时钟)：8

实例：

```
POP   SS
汇编程序产生                    0 0 0 1 0 1 1 1
POP   DS
汇编程序产生                    0 0 0 1 1 1 1 1
```

存储器/寄存器操作数：

1 0 0 0 1 1 1 1	mod 0 0 r/m

DEST = EA

定时(时钟)：存储器 17+EA
寄存器 8

实例：

```
                                POP   ALPHA
汇编产生的目标代码为：1 0 0 0 1 1 1 1  0 0 0 0 0 1 1 0 ALPHA addr-lo
ALPHA addr-hi
```

```
汇编产生的目标代码为：          POP   ALPHA [BX]
1 0 0 0 1 1 1 1  1 0 0 0 0 1 1 1  ALPHA addr-lo
ALPHA addr-hi
```

标志：不影响

说明：POP指令将SP指向的栈单元中的一个字传送到目的操作数中去，然后将SP加2。

POPF

(Pop flags off stack) ——标志退栈指令

操作：

Flags ← ((SP) + 1:(SP))
(SP) ← (SP) + 2

将栈顶字中适当的位填入标志寄存器中去，如下所示：

溢出标志	OF——位11
方向标志	DF——位10
中断标志	IF ——位 9
陷阱标志	TF——位 8
符号标志	SF——位 7
0 标志	ZF——位 6
辅助进位标志	AF——位 4
奇偶性标志	PF——位 2
进位标志	CF——位 0

然后SP再加 2 修正。

编码：

1 0 0 1 1 1 0 1

定时(时钟)： 8

实例： POPF

标志： 影响所有的标志。

说明： POPF 指令将SP指向的栈单元中的内容（16位）传送到标志寄存器中去，然后SP加 2 。

PUSH

(Push word onto stack) ——字进栈指令

操作：

1) 栈指示器 (SP) 减 2

(SP) ← (SP)−2

2) 将指令中指定的操作数内容存入 SP 指向的栈顶单元中去，SP 的内容为相对于 SS 基地址的偏移地址。

((SP+1):(SP)) ← (SRC)

标志： 不影响

编码： 有以下三种格式

寄存器操作数：(字)

0 1 0 1 0 reg

定时(时钟)： 10

实例：

```
PUSH   AX(产生目标代码为0 1 0 1 0 0 0 0)
PUSH   SI (产生目标代码为0 1 0 1 0 1 1 0)
```

段寄存器：

0 0 0 reg 1 1 0

定时(时钟)： 10

实例：

PUSH SS(产生目标代码为0 0 0 1 0 1 1 0)
PUSH ES(产生目标代码为0 0 0 0 0 1 1 0)
PUSH ES
注意：PUSH CS是合法的

存储器/寄存器操作数：

1 1 1 1 1 1 1 1	mod 1 1 0 r/m

定时(时钟)：存储器 16+EA
寄存器 10

实例：

PUSH BETA
1 1 1 1 1 1 1 1 00 110 110 汇编后产生的目标代码如下

PUSH BETA [BX]

1 1 1 1 1 1 1 1 10 110 111 汇编后产生的目标代码如下

PUSH BETA [BX] [DI]

1 1 1 1 1 1 1 1 10 110 001 Beta addr-lo Beta addr-hi

标志：不影响

说明：PUSH指令将SP 栈指示器减2，然后把源操作数中的一个字传送到当前栈指示器SP指定的单元中去。

PUSHF

(Push flags onto stack)——标志进栈指令

操作： 栈指示器SP减2，并且将标志寄存器中的内容（16位）传送到SP指定的栈单元中去。

(SP) ← (SP)−2
((SP)+1:(SP)) ← Flags

编码：

10011100

定时（时钟）： 10

实例： PUSHF

标志： 不影响

说明： PUSHF把SP寄存器减2，并把标志寄存器的内容送进SP指定的字操作数（即栈单元）中去。

RCL
(Rotate left through carry) ——带进位的循环左移指令

操作： RCL指令将指定的目的操作数（左边的）连同进位一起左移若干（COUNT）位。如果要求左移一位，则COUNT操作数应为绝对值1。或者还可在CL中存放一个数来指定左移的次数。

RCL指令使循环一直继续下去，直到COUNT消耗完（即减到0）为止。CF（进位）标志被当作为目的操作数的一部分，也就是说，CF中原有的值从低位移入目标操作数，而从目标操作数左边的高位移出的值移入CF标志。

如果COUNT为1，且初始的目的操作数值的最高两位

不相等（一个为0，一个为1），则OF（溢出）标志置位。若最高两位相等（均为0或者为1），则OF（溢出）标志复位。

如果COUNT不为1，OF（溢出）标志不确定。

```
  (temp) ← COUNT
do while (temp) ≠ 0
  (tmpcf) ← (CF)
  (CF) ← high-order bit of (EA)
  (EA) ← (EA) * 2 + (tmpcf)
  (temp) ← (temp)-1
if COUNT = 1 then
  if high-order bit of (EA) ≠ (CF) then (OF) ← 1
  else (OF) ← 0
else (OF) undefined
```

编码：

1 1 0 1 0 0 v w	mod 0 1 0 r/m

```
if v = 0 then COUNT = 1
else COUNT = (CL)
```

定时（时钟）：

(a) 一位寄存器	2
(b) 一位存储器	15+EA
(c) 可变位寄存器	8+4/bit
(d) 可变位存储器	20+EA+4/bit

实例：

```
(a) RCL   AH,   1
    RCL   BL,   1
    RCL   CX,   1
    VAL_ONE   EQU   1
    RCL   DX,   VAL_ONE
    RCL   SI,   VAL_ONE

(b) RCL   MEM_BYTE,   1
    RCL   ALPHA [DI], VAL_ONE
```

```
(c) MOV   CL,  3
    RCL   DH,  CL   ;向左循环移位 3 次
    RCL   AX,  CL

(d) MOV   CL,  6
    RCL   MEM_WORD,  CL   ;循环移位 6 次
    RCL   GANDALF_BYTE,  CL
    RCL   BETA [BX] [DI],  CL
```

标志： 影响　　CF, OF

说明： RCL指令将目的操作数连同进位标志CF一起循环左移COUNT位，参看ROL。

RCR
(Rotate right through carry) ——带进位的循环右移指令

操作： 将指定的目标操作数（左边的）连同进位标志 CF一起循环左移COUNT 次。若要循环右移一次，指定的次数要为绝对值 1，或者将合适的循环右移次数取入CL，然后由CI 控制移几位。

循环右移一直进行下去，直到COUNT 消耗完（减到 0）为止。CF 中 原先的值从最高位移入目的操作数，目的操作数的最低位移入CF。

如果COUNT为 1，且目的操作数中最高两位的值不相等(一个为 0，一个为 1)，则将溢出标志OF置位。如果最高两位相等，OF被复位。若COUNT不是 1，则OF不确定。

```
  (temp) ← COUNT
do while (temp) ≠ 0
  (tmpcf) ← (CF)
  (CF) ← low-order bit of (EA)
  (EA) ← (EA) / 2
  high-order bit of (EA) ← (tmpcf)
```

```
    (temp) ← (temp)−1
  if COUNT = 1 then
    if high-order bit of (EA) ≠ next-to-high-order bit of (EA)
      then (OF) ← 1
    else (OF) ← 0
  else (OF) undefined
```

编码：

1 1 0 1 0 0 v w	mod 0 1 1r/m

```
if v = 0 then COUNT = 1
else COUNT = (CL)
```

定时(时钟)：

(a)	一位寄存器	2
(b)	一位存储器	15+EA
(c)	可变寄存器	8+4/bit
(d)	可变存储器	20+EA+4/bit

实例：

```
(a) RCR   AH,   1
    RCR   BL,   1
    RCR   CX,   1
    VAL_ONE   EQU   1
    RCR   DX,   VAL_ONE
    RCR   SI,   VAL_ONE

(b) RCR   MEM_BYTE,   1
    RCR   ALPHA [DI], VAL_ONE

(c) MOV   CL,   3
    RCR   DH,   CL   ;向右循环移位 3 次
    RCR   AX,   CL

(d) MOV   CL,   6
    RCR   MEM_WORD,   CL   ;循环右移 6 次
    RCR   GANDALF_BYTE,   CL
    RCR   BETA [BX] [DI],   CL
```

标志： 影响　　CF, OF

说明： RCR将EA操作数中的内容通过进位标志 CF 循环右移COUNT位，参看ROR。

REP/REPE/REPZ/REPNE/REPNZ (Repeat string operation) ——重复或迭代前缀

操作： REP使串操作重复地执行，每执行一次CX内容减 1，直到CX内容减到 0 为止。

当零标志ZF的值与指令字节中第 0 位的值不相等时，比较串指令 CMPS 和扫描串指令 SCAS 将退出循环。

当 (CX) ≠ 0 时，作

对挂起的中断（若有的话）进行服务，在后继的字节中执行基本串操作 (CX) ← (CX) −1

若基本串操作指令为CMPS或SCAS，且 (ZF) ≠ z，则退出此当型循环。

编码：

1 1 1 1 0 0 1 z

定时(时钟)： 每次循环需 6 个时钟

实例：

```
1) REP   MOVS   DEST,  SOURCE        ; 参看MOVS

2) REPE  CMPS   DEST,  SOURCE
;只有当(ZF)=1时，才能在(CX)=0以前退出循
;环，也就是只有当(DI)指定的字节与(SI)指定的
;字节相等时，才能退出循环，参看CMPS。
```

3) REPNZ SCAS DEST ; 参看 SCAS
;只有当 (ZF)=1 时，即 (AL) = DEST，才能在((CX)
;= 0 以前退出循环

4) REPNZ (nonzero) = REPNE (not equal)
REPNZ (zero) = REPE (equal)

标志： 参看每条串操作指令

说明： 当((CX) 不为 0 时，REP （重复或迭代）前缀使其后面跟着的基本串操作指令重复地执行。对于 CMPS 和 SCAS 基本串操作指令来说，在每次基本串操作迭代以后，若 ZF 标志与重复前缀中的“Z”位不等，那么迭代将结束。

重复前缀可以和段修改前缀，和/或 LOCK 前缀联在一起使用，在存在多个前缀的情况下，必需禁止中断，因为从中断返回时，只能返回到指令前面的第一个前缀。

RET
(Return from procedure) ——返回指令

操作： 把栈顶存放的字置入指令指示器 IP（SP 中为栈顶的偏移地址），SP 加 2，对于段间的返回，再把栈顶的字置入代码段寄存器CS，SP 再次加 2 修正，如果在 RET 语句中指定了一个立即数，则把这个数加到SP中去。

(IP) ← ((SP) + 1:(SP))
(SP) ← (SP) + 2
若为段间的返回，则
(CS) ← ((SP) + 1:(SP))
(SP) ← (SP) + 2
若加立即数到栈指示器，则 (SP) + data

编码： 有四种格式

段内返回：

1 1 0 0 0 0 1 1

定时（时钟）： 8

实例： RET

段内返回并加立即数到栈指示器：

1 1 0 0 0 0 1 0	data-low	data-high

定时(时钟)： 12

实例：

```
RET  4
RET  12
;RET指令后的这些数字用来废除以前存入栈中的2个和6
;个参数字。由于绝大多数的栈操作都是对字进行的，所以
;RET后跟的立即数通常总是偶数（每个字要占用2个字
;节）。
```

段间返回：

1 1 0 0 1 0 1 1

定时(时钟)： 18

实例： RET

段间返回并加立即数到栈指示器SP中去：

1 1 0 0 1 0 1 0	data low	data high

定时(时钟)：17

实例：

```
RET  2  ;段间返回，先恢复IP，再恢复CS
RET  8
```

标志：不影响

说明：RET指令将控制传给由前面CALL指令保存进栈的返回地址处的指令，并可在RET之后带任意一个立即常数(偶数)，把此立即常数加到栈指示器SP中去，以便废除栈中的参数。如果是一次段间的返回（RET），即它是在标明FAR的过程情况下汇编的，则这条指令将用栈顶的两个字取代IP和CS的内容，否则，只用栈顶的一个字，取代IP的内容。

当利用间接调用（CALL）时，程序员必须注意保证CALL的类型与RET的类型是一致的，即：

```
CALL WORD PTR [BX]
```

不能调用一个FAR过程，而

```
CALL DWORD PTR [BX]
```

不能调用一个NEAR过程

ROL

(Rotate left) ——循环左移指令

操作：将指定的目的（左边的）操作数循环左移COUNT次。每次移动都是：目的操作数的最高位移入进位标志CF，而CF原有的值丢失；目的操作数中所有的位向左移一位，例如第3

位的值被第 2 位的值代替，目的操作数中空出的第 0 位用新 CF 值（即目的操作数移入的老最高位的值）填入。

循环左移一直进行下去，直到(COUNT)减到 0 为止。如果COUNT为 1，且OF的新值不等于新的最高位的值，则溢出标志OF置位；若(CF)等于最高位的值，则 OF 变为 0。但是，当COUNT不为 1 时，OF标志是不确定的。

```
(temp) ← COUNT
do while (temp) ≠ 0
  (CF) ← high-order bit of (EA)
  (EA) ← (EA) * 2 + (CF)
  (temp) ← (temp)–1

  if high-order bit of (EA) ≠ (CF) then (OF) ← 1
  else (OF) ← 0
else (OF) undefined
```

编码：

1 1 0 1 0 0 v w	mod 0 0 0 r/m

```
if v = 0 then COUNT = 1
else COUNT = (CL)
```

定时(时钟)：

(a) 一位寄存器	2
(b) 一位存储器	15+EA
(c) 可变寄存器	8+4/bit
(d) 可变存储器	20+EA+4/ bit

实例：

```
(a) ROL   AH,  1
    ROL   BL,  1
    ROL   CX,  1
    VAL_ONE  EQU  1
```

```
        ROL   DX,   VAL_ONE
        ROL   SI,   VAL_ONE

(b)     ROL   MEM_BYTE,   1
        ROL   ALPHA [DI], VAL_ONE

(c)     MOV   CL,   3
        ROL   DH,   CL   ;向左循环移位 3 次
        ROL   AX,   CL

(d)     MOV   CL,   6
        ROL   MEM_WORD,   CL   ;左循环移位 6 次
        ROL   GANDALF_BYTE,   CL
        ROL   BETA [BX] [DI],   CL
```

标志：影响　　CF, OF

说明：RCL 指令使操作数循环左移COUNT次，参看ROL。

ROR

(Rotate right) ——循环右移指令

操作：该指令使得指定的目的操作数循环右移 COUNT 次。用目的操作数中最低位的值置换CF的值，CF中原有的值丢失。目的操作数中所有的其它位向右移动，如第 2 位的值被第 3 位的取代。目的操作数中空出的最高位用 CF 的新值（目的操作数第 0 位的老值）填入。

循环右移操作一直进行下去，每循环一次COUNT 的值减 1，直到COUNT减到 0 为止。如果COUNT 为 1，且新的最高位的值不等于老的最高位的值，则将溢出标志 OF 置位；如果它们相等，则使 (OF) = 0。但是，若 COUNT 不为 1，则OF标志的状态不确定。

```
(temp) ← COUNT
DO WHILE (temp) ≠ 0
  (CF) ← low-order bit of (EA)
  (EA) ← (EA) / 2
  high-order bit of (EA) ← (CF)
  (temp) ← (temp)−1
if COUNT = 1 then
  if high-order bit of (EA) ≠ next-to-high-order bit of (EA)
    then (OF) ← 1
  else (OF) ← 0
else (OF) undefined
```

编码：

1 1 0 1 0 0 v w	mod 0 0 1 r/m

```
if v = 0 then COUNT = 1
else COUNT = (CL)
```

定时(时钟)：

(a) 一位寄存器	2
(b) 一位存储器	15+EA
(c) 可变寄存器	8+4/bit
(d) 可变存储器	20+EA+4/bit

实例：

```
(a) ROR   AH,  1
    ROR   BL,  1
    ROR   CX,  1
    VAL_ONE  EQU  1
    ROR   DX,  VAL_ONE
    ROR   SI,  VAL_ONE

(b) ROR   MEM_BYTE,  1
    ROR   ALPHA [DI], VAL_ONE

(c) MOV   CL,  3
    ROR   DH,  CL   ;循环右移 3 次
    ROR   AX,  CL
```

```
(d) MOV   CL,  6
    ROR   MEM_WORD,  CL  ;循环右移6次
    ROR   GANDALF_BYTE,  CL
    ROR   BETA [BX] [DI],  CL
```

标志： 影响　　CF, OF

说明： ROR指令将目的操作数向右移动 COUNT 次。参看RCR。

SAHF ——存AH到标志寄存器指令

操作： 将累加器高位字节AH的内容写到标志寄存器的 7～0 位中去，即用AH中的相应位去置换SF、ZF、AF、PF和CF标志。

(SF) ← AH的第 7 位
(ZF) ← AH的第 6 位
(AF) ← AH的第 4 位
(PF) ← AH的第 2 位
(CF) ← AH的第 0 位

(SF):(ZF):X:(AF):X:(PF):X:(CF) ← (AH)

编码：

10011110

定时(时钟)： 4

实例：　　SAHF

标志： 影响　　AF, CF, PF, SF, ZF

说明： SAHF指令将AH寄存器中指定的位送入标志寄存器中的SF、ZF、AF、PF和CF位中去，示有X的位可忽略。

SAL and SHL

(Shift logical left and shift arithmetic left)

——算术左移和逻辑左移指令

操作： 将指定的目的（左边的）操作数左移COUNT次。它的最高位移入进位标志CF中去，而CF中原有的值丢失。目的操作数中所有的其它位每次均向左移一位，如第3位的值被第2位的值取代，空出的最低位中填0。

移位操作一直进行到将COUNT值耗尽为止。若COUNT为1，且CF的新值不等于最高位的新值，则将溢出标志置位；若CF的新值等于最高位的新值，则OF标志被复为0。但是COUNT不是1，那么OF标志是不确定的，其值不可靠。

```
(temp) ← COUNT
do while (temp) ≠ 0
   (CF) ← high-order bit of (EA)
   (EA) ← (EA) * 2
   (temp) ← (temp)−1
if COUNT = 1 then
   if high-order bit of (EA) ≠ (CF) then (OF) ← 1
   else (OF) ← 0
else (OF) undefined
```

编码：

1 1 0 1 0 0 v w	mod 1 0 0 r/m

```
if v = 0 then COUNT = 1
else COUNT = (CL)
```

定时(时钟)：

(a) 一位寄存器	2
(b) 一位存储器	15+EA

(c) 可变寄存器	8+4/bit
(d) 可变存储器	20+EA+4/bit

实例：

```
(a) SHL  AH,  1
    SHL  BL,  1
    SHL  CX,  1
    VAL__ONE  EQU  1
    SHL  DX,  VAL__ONE
    SHL  SI,  VAL__ONE

(b) SHL  MEM__BYTE,  1
    SHL  ALPHA [DI], VAL__ONE

(c) MOV  CL,  3
    SHL  DH,  CL  ;左移多次
    SHL  AX,  CL

(d) MOV  CL,  6
    SHL  MEM__WORD,  CL  ;左移 6 次
    SHL  GANDALF__BYTE,  CL
    SHL  BETA [BX] [DI],  CL
```

标志： 影响　CF, OF, PF, SF, ZF
不确定　AF

说明： SAL（算术左移）和 SHL（逻辑左移）指令将操作数左移 COUNT 次。移位后空出的最低位中填 0。

SAR
(Shift arithmetic right)——算术右移指令

操作： 将指定的操作数（左边的）向右移 COUNT 次，其最低位移入 CF 标志，而 CF 中原有的值丢失。目的操作数中其它的位依次向右移一次，如第 2 位的值被第 3 位的值取代。空出的

最高位保持原来的值不变。也就是说，若最高位的初值为0，就把0移入各位中去。如果最高位的初值为1，那么在连续右移时就把1移入各位中去。算术右移一直进行下去，直到COUNT的值耗尽为止。如果COUNT是1且最高位的值不等于次高位的值则将溢出标志OF置位；如果最高位的值等于次高位的值，则将OF复0。若COUNT不为1，那么OF是复0的。

```
(temp) ← COUNT
do while (temp) ≠ 0
   (CF) ← low-order bit of (EA)
   (EA) ← (EA) / 2, where / is equivalent to signed
      division, rounding down
(temp) ← (temp)−1
if COUNT = 1 then
   if high-order bit of (EA) ≠ next-to-high-order bit of (EA)
      then (OF) ← 1
      else (OF) ← 0
else (OF) ← 0
```

编码：

1 1 0 1 0 0 v w	mod 1 1 1 r/m

```
if v = 0 then COUNT = 1
else COUNT = (CL)
```

定时(时钟)：

(a) 一位寄存器	2
(b) 一位存储器	15+EA
(c) 可变寄存器	8+4/bit
(d) 可变存储器	20+EA+4/bit

实例：

(a) SAR AH, 1

```
        SAR   BL,  1
        SAR   CX,  1
        VAL_ONE  EQU  1
        SAR   DX,  VAL_ONE
        SAR   SI,  VAL_ONE
```

(b)
```
        SAR   MEM_BYTE,  1
        SAR   ALPHA [DI], VAL_ONE
```

(c)
```
        MOV   CL,  3
        SAR   DH,  CL   ;右移3次
        SAR   AX,  CL
```

(d)
```
        MOV   CL,  6
        SAR   MEM_WORD,  CL   ;右移6次
        SAR   GANDALF_BYTE,  CL
        SAR   BETA [BX] [DI],  CL
```

标志：影响　　CF, OF, PF, SF, ZF.
不确定　AF

说明：SAR指令将目的操作数右移COUNT次，移入的最高位等于原来的最高位（符号扩展）

SBB
(Subtract with borrow) ——带借位的减法指令

操作：从目的（左边的）操作数中减去源（右边的）操作数。如果进位标志为1，则还要从上面相减的结果中再减去1，用最后的结果去代替最初的目的操作数。

```
if (CF) = 1 then (DEST) ← (LSRC)-(RSRC)-1
else (DEST) ← (LSRC)-(RSRC)
```

编码：有以下三种格式

存储器/寄存器操作数与寄存器操作数：

0 0 0 1 1 0 d w	mod reg r/m

if d = 1 then LSRC = REG, RSRC = EA, DEST = REG
else LSRC = EA, RSRC = REG, DEST =EA

定时(时钟)：

(a) 从寄存器减去寄存器	3
(b) 从寄存器减去存储器	9+EA
(c) 从存储器减去寄存器	16+EA

实例：

```
(a) SBB   AX,   BX
    SBB   CH,   DL

(b) SBB   DX,   MEM_WORD
    SBB   DI,   ALPHA [SI]
    SBB   BL,   MEM_BYTE [DI]

(c) SBB   MEM_WORD,   AX
    SBB   MEM_BYTE [DI],   BX
    SBB   GAMMA [BX] [DI],   SI
```

从累加器中减去立即数与借位：

0 0 0 1 1 1 0 w	data	data if w=1

(a) if w = 0 then LSRC = AL, RSRC = data, DEST = AL
(b) else LSRC = AX, RSRC = data, DEST = AX

定时(时钟)： 从寄存器中减去立即数及借位 4

实例：

```
(a) SBB   AL,   4
    VAL_SIXTY   EQU   60
    SBB   AL,   VAL_SIXTY
```

```
(b) SBB   AX,   660
    SBB   AX,   VAL__SIXTY * 6
    SBB   ,6606
```

从存储器/寄存器操作数中减去立即数与借位：

1 0 0 0 0 0 s w	mod 0 1 1 r/m	data	data if s:w=01

LSRC = EA, RSRC = data, DEST = EA

定时(时钟)： : (a) 从寄存器中减去立即数和借位 4
(b) 从存储器中减去立即数和借位 17+EA

实例：

```
(a) SBB   BX,   2001
    SBB   CL,   VAL__SIXTY
    SBB   SI,   VAL__SIXTY * 9

(b) SBB   MEM__BYTE,   12
    SBB   MEM__BYTE [DI],   VAL__SIXTY
    SBB   MEM__WORD [BX],   79
    SBB   GAMMA   [DI] [BX],   1984
```

如从寄存器/存储器字中减去的是立即数字节,则在减之前要对此立即数字节进行符号扩展，成为16位的立即数。在这种情况下的指令操作码为83H(即S：W位均为1)。

标志： 影响 AF, CF, OF, PF, SF, ZF

说明： 从目的操作数中减去源作数。若发现在减时使CF置 1， 则将结果减 1 后送回目的操作数中去。

SCAS
(Scan byte string or scan word string)

——搜索字节串或字串指令

操作：从累加器的数值减去附加段中PI寻址的字符串元素，但此操作只影响标志。然后使变址寄存器DI加1（如果方向标志DF是0），或减1（如果DF = 1），对字串，DF的增/减量为2。

```
(LSRC)-(RSRC)
if (DF) = 0 then (DI) ← (DI) + DELTA
else (DI) ← (DI)-DELTA
```

编码：

1 0 1 0 1 1 1 w

if w = 0 then LSRC = AL, RSRC = (DI), DELTA = 1
else LSRC = AX, RSRC = (DI)+1:(DI), DELTA = 2

定时(时钟)：15

实例：

```
1) CLD  ; 清除 DF,  使  DI 增量
   MOV  DI,  OFFSET  DEST_BYTE_STRING
   MOV  AL,  'M'
   SCAS  DEST_BYTE_STRING

2) STD  ;置位DF,  使  DI 减量
   MOV  DI,  OFFSET  WORD_STRING
   MOV  AX,  'MD'
   SCAS  WORD_STRING
   ;在SCAS指令中指定的操作数只供汇编程序用。汇编程
   ;序用它来验证操作数的属性和所用的当前段寄存器内容的可
   ;达性。这条指令在实际操作中是用DI寄存器去指定要搜索
   ;的单元，而不使用源程序行中指定的操作数。
```

标志：影响　　AF, CF, OF, PF, SF, ZF

说明：SCAS指令从累加器AL（或AX）中减去由DI寻址的目的操作数中的字节(或字)，不回送结果，只影响标志。在重复操作中，这条指令用于在字符串中查找一个与已知数值相同或不同的值。

SHL and SAL
(Shift logical left and shift arithmetic left)
——逻辑左移和算术左移指令

操作：将指定目的（左边的）操作数左移COUNT次，它的最高位移入进位标志CF中去，而CF中原有的值丢失。目的操作数中所有的其它位每次均向左移一位，如第3位的值被第2位的值取代，空出的最低位中填0。

移位操作一直进行到将COUNT值耗尽为止。若COUNT为1，且CF的新值不等于最高位的新值，则将溢出标志置位；若CF的新值等于最高位的新值，则OF标志被复为0。但是，如果COUNT不是1，那么OF标志是不确定的，其值不可靠。

```
(temp) ← COUNT
do while (temp) ≠ 0
  (CF) ← high-order bit of (EA)
  (EA) ← (EA) * 2
  (temp) ← (temp)−1
if COUNT = 1 then
  if high-order bit of (EA) ≠ (CF) then (OF) ← 1
  else (OF) ← 0
else (OF) undefined
```

编码：

1 1 0 1 0 0 v w	mod 1 0 0 r/m

if v = 0 then COUNT = 1
else COUNT = (CL)

定时(时钟) : (a) 一位寄存器 2
(b) 一位存储器 15+EA
(c) 可变寄存器 8+4/bit
(d) 可变存储器 20+EA+4/bit

实例：

```
(a) SHL  AH,  1
    SHL  BL,  1
    SHL  CX,  1
    VAL_ONE  EQU  1
    SHL  DX,  VAL_ONE
    SHL  SI,  VAL_ONE

(b) SHL  MEM_BYTE,  1
    SHL  ALPHA [DI], VAL_ONE

(c) MOV  CL,  3
    SHL  DH,  CL  ;左移 3 次
    SHL  AX,  CL

(d) MOV  CL,  6
    SHL  MEM_WORD,  CL  ;左移 6 次
    SHL  GANDALF_BYTE,  CL
    SHL  BETA [BX] [DI],  CL
```

标志： 影响 CF, OF, PF, SF, ZF
不确定 AF

说明： SHL（逻辑左移）和SAL（算术左移）指令将目的操作数左移COUNT次，移位后空出的最低位中填0。

SHR

(Shift logical right) ——逻辑右移指令

操作： 将指定的目的（左边的）操作数右移COUNT次。目的操作数的最低位移入进位标志，从而使进位标志原来的值就丢失了。每右移一次，目的操作数中的其它位依次向右移一位，如第2位的值被第3位的值取代，目的操作数中空出的最高位中填0。

右移一直继续下去，直到COUNT值耗尽为止。如果COUNT为1，且最高位的新值不等于次高位的值，则溢出标志OF置1。如果最高位的新值与次高位的值相等，则(OF) = 0。但是，若COUNT不为1，则OF的状态不确定，其值不可靠。

```
(temp) ← COUNT
do while (temp) ≠ 0
  (CF) ← low-order bit of (EA)
  (EA) ← (EA) / 2, where / is equivalent to unsigned division
  (temp) ← (temp)−1
if COUNT = 1 then
  if high-order bit of (EA) ≠ next-to-high-order bit of (EA)
    then (OF) ← 1
  else (OF) ← 0
else (OF) undefined
```

编码：

1 1 0 1 0 0 v w	mod 1 0 1 r/m

```
if v = 0 then COUNT = 1
else COUNT = (CL)
```

定时(时钟)：

(a) 一位寄存器	2
(b) 一位存储器	15 + EA

(c) 可变寄存器	8+4/bit
(d) 可变存储器	20+EA+4/bit

实例:

```
(a) SHR   AH,  1
    SHR   BL,  1
    SHR   CX,  1
    VAL_ONE  EQU  1
    SHR   DX,  VAL_ONE
    SHR   SI,  VAL_ONE

(b) SHR   MEM_BYTE,  1
    SHR   ALPHA [DI], VAL_ONE

(c) MOV   CL,  3
    SHR   DH,  CL  ;右移3次
    SHR   AX,  CL

(d) MOV   CL,  6
    SHR   MEM_WORD,  CL  ;右移6次
    SHR   GANDALF_BYTE,  CL
    SHR   BETA [BX] [DI],  CL
```

标志: 影响 CF, OF, PF, SF, ZF
不确定 AF

说明: SHR指令将目的操作数向右移位COUNT次。每次右移一位后，往目的操作数的左端（最高位处）补入0。

STC
(Set carry flag)——进位标志置位指令

操作: 将进位标示CF置1。

(CF) ← 1

编码：

1 1 1 1 1 0 0 1

定时(时钟)： 2

实例： STC

标志： 影响 CF

说明： STC 指令用来将CF 标志置位。

STD
(Set direction flag)——方向标志置位指令

操作： 将方向标志DF置1。

(DF) ← 1

编码：

1 1 1 1 1 1 0 1

定时(时钟) 2

实例： STD ;在字符串操作中使 DI(和 SI)减量

标志： 影响 DF.

说明： STD指令用来将DF标志置 1，DF = 1 在字符串操作中使变址指示器DI及SI自动减量。

STI

(Set interrupt flag) ——中断标志置位指令

操作：将中断标志IF置位。

(IF) ← 1

编码：

1 1 1 1 1 0 1 1

定时(时钟)：2

实例：　　STI　;允许中断

标志：影响

说明：STI指令把IF(中断标志)置1，允许处理器执行完下一条指令后响应来自可屏蔽中断請求线INTR的中断請求。

STOS

(Store byte string or store word string)

——存字节串或存字串指令

操作：用AL(或AX)中的字节(或字)取代附加段中DI所指向的单元(字节或字)内容。如果方向标志DF是零，则DI增量；如果DF = 1，则DI减量。增量或减量对字节是1；对字是2。

(DEST) ← (SRC)
if (DF) = 0 then (DI) ← (DI) + DELTA
else (DI) ← (DI)−DELTA

编码：

1 0 1 0 1 0 1 w

if w = 0 then SRC = AL, DEST = (DI), DELTA = 1
else SRC = AX, DEST = (DI) + 1:(DI), DELTA = 2

定时(时钟)： 2

实例：

```
1) MOV   DI,  OFFSET  BYTE_DEST_STRING
   STOS  BYTE_DEST_STRING

2) MOV    DI  ,OFFSET  WORD_DEST
   STOS    WORD_DEST
```

标志： 不影响

说明： STOS 指令将AL(或AX)中的一个字节(或字)传送到附加段中的DI指出的单元中去，并根据DF的状态对DI作 DELTA 的修正以使DI指向串中的下一个元素。

重复操作时，该指令可用来将一个字符串中全填满某给定的值。在指令中指定的操作数只供汇编程序用于验证类型和当前段寄存器内容的可访问性。指令的实际操作只用DI来指定所要存入的单元。

SUB
(Subtract)——减法指令

操作： 从目的(左边的)操作数中减去源(右边的)操作数，并将结果存回目的操作数中。

(DEST) ← (LSRC)−(RSRC)

编码：有三种格式

存储器/寄存器操作数和寄存器操作数：

0 0 1 0 1 0 d w	mod reg r/m

if d = 1 then LSRC = REG, RSRC = EA, DEST = REG
else LSRC = EA, RSRC = REG, DEST = EA

定时(时钟)：

(a) 从寄存器中减去寄存器	3
(b) 从寄存器中减去存储器	9+EA
(c) 从存储器中减去寄存器	16+EA

实例：

```
(a) SUB   AX,   BX
    SUB   CH,   DL

(b) SUB   DX,   MEM_WORD
    SUB   DI,   ALPHA [SI]
    SUB   BL,   MEM_BYTE [DI]

(c) SUB   MEM_WORD,   AX
    SUB   MEM_BYTE [DI],   BL
    SUB   GAMMA  [BX] [DI],   SI
```

从累加器中减去立即操作数：

0 0 1 0 1 1 0 w	data	data if w=1

(a) if w = 0 then LSRC = AL, RSRC = data, DEST = AL
(b) else LSRC = AX, RSRC = data, DEST = AX

定时(时钟)：从寄存器中减去立即数 4

实例：

```
(a) SUB   AL,  4
    VAL_SIXTY  EQU  60
    SUB   AL,  VAL_SIXTY

(b) SUB   AX,  660
    SUB   AX,  VAL_SIXTY * 6
    SUB   ,6606
```

从存储器/寄存器操作数中减去立即操作数：

1 0 0 0 0 0 s w	mod 1 0 1 r/m	data	data if s:w=01

LSRC = EA, RSRC = data, DEST = EA

定时(时钟)：

(a) 从寄存器减去立即数	4
(b) 从存储器减去立即数	17+EA

实例：

```
(a) SUB   BX,  2001
    SUB   CL,VAL_SIXTY
    SUB   SI,  VAL_SIXTY * 9

(b) SUB   MEM_BYTE,  12
    SUB   MEM_BYTE [DI],  VAL_SIXTY
    SUB   MEM_WORD [BX],  79
    SUB   GAMMA [DI] [BX],  1984
```

如果从一个寄存器/存储器字中减去一个立即数字节，则在执行减法以前，先要对此字节进行符号扩展成为16位的字。在这种情况下，指令操作码字节为83H（即S：W均为1）。

标志： 影响　AF, CF, OF, PF, SF, ZF

说明： SUB 指令用来执行从目的操作中减去源（右边的）操作数并将结果送回目的操作数。

TEST

(Test, or logical compare) ——检测或逻辑比较指令

操作： TEST指令对两个操作数（目的和源）执行逻辑“与”操作，只影响标志，结果不回送。原来参加运算的两个操作数均不改变。进位和溢出标志被复位。

(LSRC) & (RSRC)
(CF) ← 0
(OF) ← 0

编码： 有三种格式

存储器/寄存器操作数与寄存器操作数：

1 0 0 0 0 1 0 w	mod reg r/m

LSRC = REG, RSRC = EA

定时(时钟)：

(a) 寄存器与寄存器	3
(b) 寄存器与存储器	9+EA

实例：

```
(a) TEST  AX,  DX
    TEST  ,DX  ;同上
    TEST  SI,  BP
    TEST  BH,  CL

(b) TEST  MEM_WORD,  SI
    TEST  MEM_BYTE,  CH
    TEST  ALPHA [DI],  DX
```

```
TEST  BETA [BX] [SI],  CX
TEST  DI,  MEM_WORD
TEST  CH,  MEM_BYTE
TEST  AX,  GAMMA [BP] [SI]
```

立即操作数与累加器：

1 0 1 0 1 0 0 w	data	data if w=1

(a) if w = 0 then LSRC = AL, RSRC = data
(b) else LSRC = AX, RSRC = data

定时(时钟)：立即数与寄存器　　4

实例：

```
TEST  AL,  6
TEST  AL,  IMM_VALUE_DRIVE11
TEST  AX,  IMM_VAL_909
TEST  ,999
TEST  AX,  999  ;同上
```

立即操作数与存储器/寄存器操作数：

1 1 1 1 0 1 1 w	mod 0 0 0 r/m	data	data if w=1

LSRC = EA, RSRC = data

定时(时钟)：
(a) 立即数与寄存器　　4
(b) 立即数与存储器　　10 + EA

实例：

```
(a) TEST  BH,  7
    TEST  CL,  19_IMM_BYTE
    TEST  DX,  IMM_DATA_WORD
    TEST  SI,  798
```

(b) TEST MEM_WORD, IMM_DATA_BYTE
TEST GAMMA [BX], IMM_BYTE
TEST [BP] [DI], 6ACEH

标志： 影响 CF, OF, PF, SF, ZF.
不确定 AF

说明： TEST 将两个操作数按位进行逻辑“与”，根据“与”的结果影响标志，但不送回结果。该指令可用来清进位和溢出标志。除去 TEST 立即数字节与存储器字的情况以外，通常要求源（右边的）操作数与目的（左边的）操作数有同样的属性，即字节或字。

WAIT

(Wait) ——等待指令

操作： 无

编码：

10011011

定时(时钟)： 3

实例： WAIT

标志： 不影响

说明： 若8086的$\overline{\text{TEST}}$引脚上的信号无效，WAIT 指令使8086处理器进入一个等待状态。等待状态可被一个“外部的中断”所中断，这时保存进栈的断点（代码单元）地址为WAIT 指令所

在的单元地址。这样就使得从中断服务返回时，CPU重新进入等待状态。

当8086的$\overline{TEST}$引脚上的信号有效时，等待状态被清除。同时，恢复执行被暂停的8086指令。在恢复执行指令时禁止外部中断，直到执行完下一条指令后才允许外部中断。

WAIT指令用来处理器与外部硬件同步。

XCHG (Exchange)——交换指令

有两种XCHG指令的格式：一种用于累加器和其它通用字寄存器交换内容；另一种用于寄存器和寄存器（或存储器）操作数交换内容。

操作：

1) 目的操作数（左边的）的内容，暂时地存入一个内部的工作寄存器

(Temp) ← (DEST)

2) 用源（右边的）操作数的内容，置换目的操作数的内容

(DEST) ← (SRC)

3) 将暂存在内部工作寄存器中的目的操作数内容移入源操作数

(SRC) ← (Temp)

标志： 不影响

编码： 有两种格式

寄存器操作数与累加器：

1 0 0 1 0 reg

SRC = REG, DEST = AX

定时(时钟)： 3

实例：

```
XCHG   AX,   BX
XCHG   SI,   AX
XCHG   CX,   AX
```

存储器/寄存器操作数与寄存器操作数：

1 0 0 0 0 1 1 w	mod reg r/m

SRC = EA, DEST = REG

定时(时钟)： 存储器与寄存器　　17+EA
寄存器与存储器　　4

实例：

```
XCHG   BETA_WORD,   CX
XCHG   BX,   DELTA_WORD
XCHG   DH,   ALPHA_BYTE
XCHG   BL,   AL
```

说明： XCHG指令将源操作数和目的操作数中的字节（或字）进行交换。XCHG指令操作数中不能用段寄存器。

XLAT
(Translate) ——换码指令

操作： 用表中的一个字节来取代累加器的内容，必须事先把表的起始地址装入BX寄存器中。AL的初始内容是所要找的表中的字节，距离表的起始地址的字节数。

(AL) ← ((BX) + (AL))

编码：

1 1 0 1 0 1 1 1

定时(时钟)： 11

实例：
```
MOV  BX,  OFFSET  TABLE_NAME
XLAT  TABLE_ENTRY
;(参看在LODS指令中的实例)
```

标志： 不影响

说明： XLAT指令用来执行表查换码操作。AL寄存器用来作为进入一个基址在BX中的表（最多256字节长的表）的索引。找到的操作数字节送入AL中。

XOR
(Exclusive or)——异或指令

操作： XOR指令对两个操作数进行按位“异或”若对应位数值相等，则结果位为0。若对应位不相等，则结果应为1。

(DEST) ← (LSRC)‖(RSRC)
(CF) ← 0
(OF) ← 0

编码：有三种格式

存储器/寄存器与寄存器：

0 0 1 1 0 0 d w	mod reg r/m

if d = 1 then LSRC = REG, RSRC = EA, DEST = REG
else LSRC = EA, RSRC = REG, DEST = EA

定时(时钟)：

(a) 寄存器到寄存器	3
(b) 存储器到寄存器	9 + EA
(c) 寄存器到存储器	16 + EA

实例：

```
(a) XOR  AH,  BL      ;结果在 AH,  BL 中不变
    XOR  SI,  DX      ;结果在 SI,  DX 中不变
    XOR  CX,  DI      ;结果在 CX,  DI 中不变

(b) XOR  AX,  MEM_WORD
    XOR  CL,  MEM_BYTE [SI]
    XOR  SI,  ALPHA [BX] [SI]

(c) XOR  BETA [BX] [DI],  AX
    XOR  MEM_BYTE,  DH
    XOR  GAMMA [DI],  BX
```

立即操作数到累加器：

0 0 1 1 0 1 0 w	data	data if w=1

if w = 0 then LSRC = AL, RSRC = data, DEST = AL
else LSRC = AX, RSRC = data, DEST = AX

定时(时钟)立即数到寄存器　　4

实例：

a) XOR AL, 11110110B
XOR AL, 0F6H

b) XOR AX, 23F6H
XOR AX, 75Q
XOR ,23F6H ;AX为目的操作数

立即操作数到存储器/寄存器操作数：

1 0 0 0 0 0 0 w	mod 1 1 0 r/m	data	data if w=1

LSRC = EA, RSRC = data, DEST = EA

定时(时钟) ：立即数到寄存器 4
立即数到存储器 17 + EA

实例：

a) XOR AH, 0F6H
XOR CL, 37
XOR DI, 23F5H

b) XOR MEM_BYTE, 3DH
XOR GAMMA [BX] [DI], 0FACEH
XOR ALPHA [DI], VAL_EQUD_33H

标志： 影响 CF, OF, PF, SF, ZF.
不确定 AF

说明： XOR 指令用来实现两个操作数之间的“异或”操作并将结果送回目的操作数中去。

指令系统符号说明：

符　号	意　义	符　号	意　义
AX	累加器（16位）	Flags	存放9位标志位的16位寄存器
AH	累加器AX(高字节)	DI	目标变址寄存器（16位）
AL	累加器AX(低字节)	SI	源变址寄存器（16位）
BX	寄存器BX（16位）	CS	代码段寄存器（16位）
BH	寄存器BX高字节	DS	数据段寄存器（16位）
BL	寄存器BX低字节	ES	附加段寄存器（16位）
CX	寄存器CX（16位）	SS	堆栈段寄存器（16位）
CH	寄存器CX高字节	LSRC, RSRC	左操作数，右操作数。左操作数称为目的操作数，右为源操作数
CL	寄存器CX低字节		
DX	寄存器DX（16位）	reg	在一条指令中说明寄存器的字段
DH	寄存器DX的高字节	EA	有效地址（16位）
DL	寄存器DX的低字节	r/m	mod rm字节中的第2,1,0位用于存取存储器操作数。这三位字段连同“mode”和“w”字段来定义EA
SP	堆栈指针（16位）		
BP	基址指针（16位）		
IP	指令指针（16位）		

符　号	意　义	符　号	意　义
mode	mod rm 字节中的第7，6位这两位字段定义存储方式		作数的低8位放在寄存器BX的内容指出的存储器单元中，该操作数的高8位放在相邻存储器单元（DX）+1中
w	一条指令中的一位字段，w=0表示字节指令，w=1表示字指令	((BX)+1,(BX))	意思是这里存放一个16位操作数
d	指令中一位表示方向的字段，由它指定寄存器是源，还是目的单元	((DX)+1,(DX))	这个16位操作数的存储器单元低字节由DX指出，而高字节由相邻的存储单元指出
(…)	括号意思是寄存器或存储器单元的内容	addr	存储器内字节的地址（16位）
(BX)	表示寄存器BX的内容，它可作为存放8位操作数的地址。如果用于汇编指令，BX只用方括号括起	addr-low	一个地址的低字节
		addr-high	一个地址的高字节
		addr+1:addr	存储器中两个连续字节的地址，从addr开始
((BX))	这个意思是8位操作数，该存储单元的内容的地址由寄存器BX的内容指出。该符号不在源语句中出现	data	立即操作数（w=0时为8位；w=1时为16位）
		data-low	16位数据字的低字节
		data-high	16位数据字的高字节
		disp	位移量
(BX)+1,(BX)	是一16位操作数的地址，这个操	disp-low	16位位移量的低字节

符　号	意　义	符　号	意　义
disp-high	16位位移量的高字节	AF	辅助进位标志
←	赋值	CF	进位标志
+	加法	PF	奇偶标志
-	减法	SF	符号标志
*	乘法	ZF	零标志
/	除法	DF	指令步进方向标志（用于字符串操作）
%	模		
&	与	IF	中断允许标志
|	或（或除）	OF	溢出标志
‖	异或（或等于）	TF	陷阱——执行单步指令标志

附录 2　8086指令表

（按 16 进制码）

00	00000000	MOD REGR/M	ADD	EA,REG	BYTE ADD (REG) TO EA
01	00000001	MOD REGR/M	ADD	EA,REG	WORD ADD (REG) TO EA
02	00000010	MOD REGR/M	ADD	REG,EA	BYTE ADD (EA) TO REG
03	00000011	MOD REGR/M	ADD	REG,EA	WORD ADD (EA) TO REG
04	00000100		ADD	AL,DATA8	BYTE ADD DATA TO REG AL
05	00000101		ADD	AX,DATA16	WORD ADD DATA TO REG AX
06	00000110		PUSH	ES	PUSH (ES) ON STACK
07	00000111		POP	ES	POP STACK TO REG ES
08	00001000	MOD REGR/M	OR	EA,REG	BYTE OR (REG) TO EA
09	00001001	MOD REGR/M	OR	EA,REG	WORD OR (REG) TO EA
0A	00001010	MOD REGR/M	OR	REG,EA	BYTE OR (EA) TO REG
0B	00001011	MOD REGR/M	OR	REG,EA	WORD OR (EA) TO REG
0C	00001100		OR	AL,DATA8	BYTE OR DATA TO REG AL
0D	00001101		OR	AX,DATA16	WORD OR DATA TO REG AX
0E	00001110		PUSH	CS	PUSH (CS) ON STACK
0F	00001111		(not used)		
10	00010000	MOD REGR/M	ADC	EA,REG	BYTE ADD (REG) W/ CARRY TO EA
11	00010001	MOD REGR/M	ADC	EA,REG	WORD ADD (REG) W/ CARRY TO EA
12	00010010	MOD REGR/M	ADC	REA,EA	BYTE ADD (EA) W/ CARRY TO REG
13	00010011	MOD REGR/M	ADC	REG,EA	WORD ADD (EA) W/ CARRY TO REG
14	00010100		ADC	AL,DATA8	BYTE ADD DATA W/CARRY TO REG AL
15	00010101		ADC	AX,DATA16	WORD ADD DATA W/ CARRY TO REG AX
16	00010110		PUSH	SS	PUSH (SS) ON STACK
17	00010111		POP	SS	POP STACK TO REG SS
18	00011000	MOD REGR/M	SBB	EA,REG	BYTE SUB (REG) W/ BORROW FROM EA
19	00011001	MOD REGR/M	SBB	EA,REG	WORD SUB (REG) W/ BORROW FROM EA
1A	00011010	MOD REGR/M	SBB	REG,EA	BYTE SUB (EA) W/ BORROW FROM REG
1B	00011011	MOD REGR/M	SBB	REG,EA	WORD SUB (EA) W/ BORROW FROM REG
1C	00011100		SBB	AL,DATA8	BYTE SUB DATA W/ BORROW FROM REG AL
1D	00011101		SBB	AX,DATA16	WORD SUB DATA W/ BORROW FROM REG AX
1E	00011110		PUSH	DS	PUSH (DS) ON STACK
1F	00011111		POP	DS	POP STACK TO REG DS
20	00100000	MOD REGR/M	AND	EA,REG	BYTE AND (REG) TO EA
21	00100001	MOD REGR/M	AND	EA,REG	WORD AND (REG) TO EA
22	00100010	MOD REGR/M	AND	REG,EA	BYTE AND (EA) TO REG
23	00100011	MOD REGR/M	AND	REG,EA	WORD AND (EA) TO REG
24	00100100		AND	AL,DATA8	BYTE AND DATA TO REG AL
25	00100101		AND	AX,DATA16	WORD AND DATA TO REG AX
26	00100110		ES:		SEGMENT OVERIDE W/ SEGMENT REG ES
27	00100111		DAA		DECIMAL ADJUST FOR ADD
28	00101000	MOD REGR/M	SUB	EA,REG	BYTE SUBTRACT (REG) FROM EA
29	00101001	MOD REGR/M	SUB	EA,REG	WORD SUBTRACT (REG) FROM EA
2A	00101010	MOD REGR/M	SUB	REG,EA	BYTE SUBTRACT (EA) FROM REG
2B	00101011	MOD REGR/M	SUB	REG,EA	WORD SUBTRACT (EA) FROM REG
2C	00101100		SUB	AL,DATA8	BYTE SUBTRACT DATA FROM REG AL
2D	00101101		SUB	AX,DATA16	WORD SUBTRACT DATA FROM REG AX
2E	00101110		CS:		SEGMENT OVERIDE W/ SEGMENT REG CS
2F	00101111		DAS		DECIMAL ADJUST FOR SUBTRACT
30	00110000	MOD REGR/M	XOR	EA,REG	BYTE XOR (REG) TO EA
31	00110001	MOD REGR/M	XOR	EA,REG	WORD XOR (REG) TO EA
32	00110010	MOD REGR/M	XOR	REG,EA	BYTE XOR (EA) TO REG
33	00110011	MOD REGR/M	XOR	REG,EA	WORD XOR (EA) TO REG
34	00110100		XOR	AL,DATA8	BYTE XOR DATA TO REG AL
35	00110101		XOR	AX,DATA16	WORD XOR DATA TO REG AX
36	00110110		SS:		SEGMENT OVERIDE W/ SEGMENT REG SS
37	00110111		AAA		ASCII ADJUST FOR ADD
38	00111000	MOD REGR/M	CMP	EA,REG	BYTE COMPARE (EA) WITH (REG)
39	00111001	MOD REGR/M	CMP	EA,REG	WORD COMPARE (EA) WITH (REG)
3A	00111010	MOD REGR/M	CMP	REG,EA	BYTE COMPARE (REG) WITH (EA)
3B	00111011	MOD REGR/M	CMP	REG,EA	WORD COMPARE (REG) WITH (EA)
3C	00111100		CMP	AL,DATA8	BYTE COMPARE DATA WITH (AL)
3D	00111101		CMP	AX,DATA16	WORD COMPARE DATA WITH (AX)

3E 00111110		DS:		SEGMENT OVERIDE W/ SEGMENT REG DS
3F 00111111		AAS		ASCII ADJUST FOR SUBTRACT
40 01000000		INC	AX	INCREMENT (AX)
41 01000001		INC	CX	INCREMENT (CX)
42 01000010		INC	DX	INCREMENT (DX)
43 01000011		INC	BX	INCREMENT (BX)
44 01000100		INC	SP	INCREMENT (SP)
45 01000101		INC	BP	INCREMENT (BP)
46 01000110		INC	SI	INCREMENT (SI)
47 01000111		INC	DI	INCREMENT (DI)
48 01001000		DEC	AX	DECREMENT (AX)
49 01001001		DEC	CX	DECREMENT (CX)
4A 01001010		DEC	DX	DECREMENT (DX)
4B 01001011		DEC	BX	DECREMENT (BX)
4C 01001100		DEC	SP	DECREMENT (SP)
4D 01001101		DEC	BP	DECREMENT (BP)
4E 01001110		DEC	SI	DECREMENT (SI)
4F 01001111		DEC	DI	DECREMENT (DI)
50 01010000		PUSH	AX	PUSH (AX) ON STACK
51 01010001		PUSH	CX	PUSH (CX) ON STACK
52 01010010		PUSH	DX	PUSH (DX) ON STACK
53 01010011		PUSH	BX	PUSH (BX) ON STACK
54 01010100		PUSH	SP	PUSH (SP) ON STACK
55 01010101		PUSH	BP	PUSH (BP) ON STACK
56 01010110		PUSH	SI	PUSH (SI) ON STACK
57 01010111		PUSH	DI	PUSH (DI) ON STACK
58 01011000		POP	AX	POP STACK TO REG AX
59 01011001		POP	CX	POP STACK TO REG CX
5A 01011010		POP	DX	POP STACK TO REG DX
5B 01011011		POP	BX	POP STACK TO REG BX
5C 01011100		POP	SP	POP STACK TO REG SP
5D 01011101		POP	BP	POP STACK TO REG BP
5E 01011110		POP	SI	POP STACK TO REG SI
5F 01011111		POP	DI	POP STACK TO REG DI
60 01100000		(not used)		
61 01100001		(not used)		
62 01100010		(not used)		
63 01100011		(not used)		
64 01100100		(not used)		
65 01100101		(not used)		
66 01100110		(not used)		
67 01100111		(not used)		
68 01101000		(not used)		
69 01101001		(not used)		
6A 01101010		(not used)		
6B 01101011		(not used)		
6C 01101100		(not used)		
6D 01101101		(not used)		
6E 01101110		(not used)		
6F 01101111		(not used)		
70 01110000		JO	DISP8	JUMP ON OVERFLOW
71 01110001		JNO	DISP8	JUMP ON NOT OVERFLOW
72 01110010		JB/JNAE	DISP8	JUMP ON BELOW/NOT ABOVE OR EQUAL
73 01110011		JNB/JAE	DISP8	JUMP ON NOT BELOW/ABOVE OR EQUAL
74 01110100		JE/JZ	DISP8	JUMP ON EQUAL/ZERO
75 01110101		JNE/JNZ	DISP8	JUMP ON NOT EQUAL/NOT ZERO
76 01110110		JBE/JNA	DISP8	JUMP ON BELOW OR EQUAL/NOT ABOVE
77 01110111		JNBE/JA	DISP8	JUMP ON NOT BELOW OR EQUAL/ABOVE
78 01111000		JS	DISP8	JUMP ON SIGN
79 01111001		JNS	DISP8	JUMP ON NOT SIGN
7A 01111010		JP/JPE	DISP8	JUMP ON PARITY/PARITY EVEN
7B 01111011		JNP/JPO	DISP8	JUMP ON NOT PARITY/PARITY ODD
7C 01111100		JL/JNGE	DISP8	JUMP ON LESS/NOT GREATER OR EQUAL
7D 01111101		JNL/JGE	DISP8	JUMP ON NOT LESS/GREATER OR EQUAL
7E 01111110		JLE/JNG	DISP8	JUMP ON LESS OR EQUAL/NOT GREATER
7F 01111111		JNLE/JG	DISP8	JUMP ON NOT LESS OR EQUAL/GREATER
80 10000000	MOD 000 R/M	ADD	EA,DATA8	BYTE ADD DATA TO EA
80 10000000	MOD 001 R/M	OR	EA,DATA8	BYTE OR DATA TO EA
80 10000000	MOD 010 R/M	ADC	EA,DATA8	BYTE ADD DATA W/ CARRY TO EA
80 10000000	MOD 011 R/M	SBB	EA,DATA8	BYTE SUB DATA W/ BORROW FROM EA
80 10000000	MOD 100 R/M	AND	EA,DATA8	BYTE AND DATA TO EA

80 10000000	MOD 101 R/M	SUB	EA,DATA8	BYTE SUBTRACT DATA FROM EA
80 10000000	MOD 110 R/M	XOR	EA,DATA8	BYTE XOR DATA TO EA
80 10000000	MOD 111 R/M	CMP	EA,DATA8	BYTE COMPARE DATA WITH (EA)
81 10000001	MOD 000 R/M	ADD	EA,DATA16	WORD ADD DATA TO EA
81 10000001	MOD 001 R/M	OR	EA,DATA16	WORD OR DATA TO EA
81 10000001	MOD 010 R/M	ADC	EA,DATA16	WORD ADD DATA W/ CARRY TO EA
81 10000001	MOD 011 R/M	SBB	EA,DATA16	WORD SUB DATA W/ BORROW FROM EA
85 10000001	MOD 100 R/M	AND	EA,DATA16	WORD AND DATA TO EA
81 10000001	MOD 101 R/M	SUB	EA,DATA16	WORD SUBTRACT DATA FROM EA
81 10000001	MOD 110 R/M	XOR	EA,DATA16	WORD XOR DATA TO EA
81 10000001	MOD 111 R/M	CMP	EA,DATA16	WORD COMPARE DATA WITH (EA)
82 10000010	MOD 000 R/M	ADD	EA,DATA8	BYTE ADD DATA TO EA
82 10000010	MOD 001 R/M	(not used)		
82 10000010	MOD 010 R/M	ADC	EA,DATA8	BYTE ADD DATA W/ CARRY TO EA
82 10000010	MOD 011 R/M	SBB	EA,DATA8	BYTE SUB DATA W/ BORROW FROM EA
82 10000010	MOD 100 R/M	(not used)		
82 10000010	MOD 101 R/M	SUB	EA,DATA8	BYTE SUBTRACT DATA FROM EA
82 10000010	MOD 110 R/M	(not used)		
82 10000010	MOD 111 R/M	CMP	EA,DATA8	BYTE COMPARE DATA WITH (EA)
83 10000011	MOD 000 R/M	ADD	EA,DATA8	WORD ADD DATA TO EA
83 10000011	MOD 001 R/M	(not used)		
83 10000011	MOD 010 R/M	ADC	EA,DATA8	WORD ADD DATA W/ CARRY TO EA
83 10000011	MOD 011 R/M	SBB	EA,DATA8	WORD SUB DATA W/ BORROW FROM EA
83 10000011	MOD 100 R/M	(not used)		
83 10000011	MOD 101 R/M	SUB	EA,DATA8	WORD SUBTRACT DATA FROM EA
83 10000011	MOD 110 R/M	(not used)		
83 10000011	MOD 111 R/M	CMP	EA,DATA8	WORD COMPARE DATA WITH (EA)
84 10000100	MOD REGR/M	TEST	EA,REG	BYTE TEST (EA) WITH (REG)
85 10000101	MOD REGR/M	TEST	EA,REG	WORD TEST (EA) WITH (REG)
86 10000110	MOD REGR/M	XCHG	REG,EA	BYTE EXCHANGE (REG) WITH (EA)
87 10000111	MOD REGR/M	XCHG	REG,EA	WORD EXCHANGE (REG) WITH (EA)
88 10001000	MOD REGR/M	MOV	EA,REG	BYTE MOVE (REG) TO EA
89 10001001	MOD REGR/M	MOV	EA,REG	WORD MOVE (REG) TO EA
8A 10001010	MOD REGR/M	MOV	REG,EA	BYTE MOVE (EA) TO REG
8B 10001011	MOD REGR/M	MOV	REG,EA	WORD MOVE (EA) TO REG
8C 10001100	MOD 0SR R/M	MOV	EA,SR	WORD MOVE (SEGMENT REG SR) TO EA
8C 10001100	MOD 1-- R/M	(not used)		
8D 10001101	MOD REGR/M	LEA	REG,EA	LOAD EFFECTIVE ADDRESS OF EA TO REG
8E 10001110	MOD 0SR R/M	MOV	SR,EA	WORD MOVE (EA) TO SEGMENT REG SR
8E 10001110	MOD -- R/M	(not used)		
8F 10001111	MOD 000 R/M	POP	EA	POP STACK TO EA
8F 10001111	MOD 001 R/M	(not used)		
8F 10001111	MOD 010 R/M	(not used)		
8F 10001111	MOD 011 R/M	(not used)		
8F 10001111	MOD 100 R/M	(not used)		
8F 10001111	MOD 101 R/M	(not used)		
8F 10001111	MOD 110 R/M	(not used)		
8F 10001111	MOD 111 R/M	(not used)		
90 10010000		XCHG	AX,AX	EXCHANGE (AX) WITH (AX), (NOP)
91 10010001		XCHG	AX,CX	EXCHANGE (AX) WITH (CX)
92 10010010		XCHG	AX,DX	EXCHANGE (AX) WITH (DX)
93 10010011		XCHG	AX,BX	EXCHANGE (AX) WITH (BX)
94 10010100		XCHG	AX,SP	EXCHANGE (AX) WITH (SP)
95 10010101		XCHG	AX,BP	EXCHANGE (AX) WITH (BP)
96 10010110		XCHG	AX,SI	EXCHANGE (AX) WITH (SI)
97 10010111		XCHG	AX,DI	EXCHANGE (AX) WITH (DI)
98 10011000		CBW		BYTE CONVERT (AL) TO WORD (AX)
99 10011001		CWD		WORD CONVERT (AX) TO DOUBLE WORD
9A 10011010		CALL	DISP16,SEG16	DIRECT INTER SEGMENT CALL
9B 10011011		WAIT		WAIT FOR TEST SIGNAL
9C 10011100		PUSHF		PUSH FLAGS ON STACK
9D 10011101		POPF		POP STACK TO FLAGS
9E 10011110		SAHF		STORE (AH) INTO FLAGS
9F 10011111		LAHF		LOAD REG AH WITH FLAGS
A0 10100000		MOV	AL,ADDR16	BYTE MOVE (ADDR) TO REG AL
A1 10100001		MOV	AX,ADDR16	WORD MOVE (ADDR) TO REG AX
A2 10100010		MOV	ADDR16,AL	BYTE MOVE (AL) TO ADDR
A3 10100011		MOV	ADDR16,AX	WORD MOVE (AX) TO ADDR
A4 10100100		MOVS	DST8SRC8	BYTE MOVE, STRING OP
A5 10100101		MOVS	DST16,SRC16	WORD MOVE, STRING OP
A6 10100110		CMPS	SIPTR,DIPTR	COMPARE BYTE, STRING OP

A7 10100111			CMPS	SIPTR,DIPTR	COMPARE WORD, STRING OP
A8 10101000			TEST	AL,DATA8	BYTE TEST (AL) WITH DATA
A9 10101001			TEST	AX,DATA16	WORD TEST (AX) WITH DATA
AA10101010			STOS	DST8	BYTE STORE, STRING OP
AB10101011			STOS	DST16	WORD STORE, STRING OP
AC10101100			LODS	SRC8	BYTE LOAD, STRING OP
AD10101101			LODS	SRC16	WORD LOAD, STRING OP
AE10101110			SCAS	DIPTR8	BYTE SCAN, STRING OP
AF10101111			SCAS	DIPTR16	WORD SCAN, STRING OP
B0 10110000			MOV	AL,DATA8	BYTE MOVE DATA TO REG AL
B1 10110001			MOV	CL,DATA8	BYTE MOVE DATA TO REG CL
B2 10110010			MOV	DL,DATA8	BYTE MOVE DATA TO REG DL
B3 10110011			MOV	BL,DATA8	BYTE MOVE DATA TO REG BL
B4 10110100			MOV	AH,DATA8	BYTE MOVE DATA TO REG AH
B5 10110101			MOV	CH,DATA8	BYTE MOVE DATA TO REG CH
B6 10110110			MOV	DH,DATA8	BYTE MOVE DATA TO REG DH
B7 10110111			MOV	BH,DATA8	BYTE MOVE DATA TO REG BH
B8 10111000			MOV	AX,DATA16	WORD MOVE DATA TO REG AX
B9 10111001			MOV	CX,DATA16	WORD MOVE DATA TO REG CX
BA10111010			MOV	DX,DATA16	WORD MOVE DATA TO REG DX
BB10111011			MOV	BX,DATA16	WORD MOVE DATA TO REG BX
BC10111100			MOV	SP,DATA16	WORD MOVE DATA TO REG SP
BD10111101			MOV	BP,DATA16	WORD MOVE DATA TO REG BP
BE10111110			MOV	SI,DATA16	WORD MOVE DATA TO REG SI
BF10111111			MOV	DI,DATA16	WORD MOVE DATA TO REG DI
C0 11000000			(not used)		
C1 11000001			(not used)		
C2 11000010			RET	DATA16	INTRA SEGMENT RETURN, ADD DATA TO REG SP
C3 11000011			RET		INTRA SEGMENT RETURN
C4 11000100	MOD REGR/M		LES	REG,EA	WORD LOAD REG AND SEGMENT REG ES
C5 11000101	MOD REGR/M		LDS	REG,EA	WORD LOAD REG AND SEGMENT REG DS
C6 11000110	MOD 000	R/M	MOV	EA,DATA8	BYTE MOVE DATA TO EA
C6 11000110	MOD 001	R/M	(not used)		
C6 11000110	MOD 010	R/M	(not used)		
C6 11000110	MOD 011	R/M	(not used)		
C6 11000110	MOD 100	R/M	(not used)		
C6 11000110	MOD 101	R/M	(not used)		
C6 11000110	MOD 110	R/M	(not used)		
C6 11000110	MOD 111	R/M	(not used)		
C7 11000111	MOD 000	R/M	MOV	EA,DATA16	WORD MOVE DATA TO EA
C7 11000111	MOD 001	R/M	(not used)		
C7 11000111	MOD 010	R/M	(not used)		
C7 11000111	MOD 011	R/M	(not used)		
C7 11000111	MOD 100	R/M	(not used)		
C7 11000111	MOD 101	R/M	(not used)		
C7 11000111	MOD 110	R/M	(not used)		
C7 11000111	MOD 111	R/M	(not used)		
C8 11001000			(not used)		
C9 11001001			(not used)		
CA11001010			RET	DATA16	INTER SEGMENT RETURN, ADD DATA TO REG SP
CB11001011			RET		INTER SEGMENT RETURN
CC11001100			INT	3	TYPE 3 INTERRUPT
CD11001101			INT	TYPE	TYPED INTERRUPT
CE11001110			INTO		INTERRUPT ON OVERFLOW
CF11001111			IRET		RETURN FROM INTERRUPT
D0 11010000	MOD 000	R/M	ROL	EA,1	BYTE ROTATE EA LEFT 1 BIT
D0 11010000	MOD 001	R/M	ROR	EA,1	BYTE ROTATE EA RIGHT 1 BIT
D0 11010000	MOD 010	R/M	RCL	EA,1	BYTE ROTATE EA LEFT THRU CARRY 1 BIT
D0 11010000	MOD 011	R/M	RCR	EA,1	BYTE ROTATE EA RIGHT THRU CARRY 1 BIT
D0 11010000	MOD 100	R/M	SHL	EA,1	BYTE SHIFT EA LEFT 1 BIT
D0 11010000	MOD 101	R/M	SHR	EA,1	BYTE SHIFT EA RIGHT 1 BIT
D0 11010000	MOD 110	R/M	(not used)		
D0 11010000	MOD 111	R/M	SAR	EA,1	BYTE SHIFT SIGNED EA RIGHT 1 BIT
D1 11010001	MOD 000	R/M	ROL	EA,1	WORD ROTATE EA LEFT 1 BIT
D1 11010001	MOD 001	R/M	ROR	EA,1	WORD ROTATE EA RIGHT 1 BIT
D1 11010001	MOD 010	R/M	RCL	EA,1	WORD ROTATE EA LEFT THRU CARRY 1 BIT
D1 11010001	MOD 011	R/M	RCR	EA,1	WORD ROTATE EA RIGHT THRU CARRY 1 BIT
D1 11010001	MOD 100	R/M	SHL	EA,1	WORD SHIFT EA LEFT 1 BIT
D1 11010001	MOD 101	R/M	SHR	EA,1	WORD SHIFT EA RIGHT 1 BIT
D1 11010001	MOD 110	R/M	(not used)		
D1 11010001	MOD 111	R/M	SAR	EA,1	WORD SHIFT SIGNED EA RIGHT 1 BIT

D2 11010010	MOD 000 R/M	ROL	EA,CL	BYTE ROTATE EA LEFT (CL) BITS
D2 11010010	MOD 001 R/M	ROR	EA,CL	BYTE ROTATE EA RIGHT (CL) BITS
D2 11010010	MOD 010 R/M	RCL	EA,CL	BYTE ROTATE EA LEFT THRU CARRY (CL) BITS
D2 11010010	MOD 011 R/M	RCR	EA,CL	BYTE ROTATE EA RIGHT THRU CARRY (CL) BITS
D2 11010010	MOD 100 R/M	SHL	EA,CL	BYTE SHIFT EA LEFT (CL) BITS
D2 11010010	MOD 101 R/M	SHR	EA,CL	BYTE SHIFT EA RIGHT (CL) BITS
D2 11010010	MOD 110 R/M	(not used)		
D2 11010010	MOD 111 R/M	SAR	EA,CL	BYTE SHIFT SIGNED EA RIGHT (CL) BITS
D3 11010011	MOD 000 R/M	ROL	EA,CL	WORD ROTATE EA LEFT (CL) BITS
D3 11010011	MOD 001 R/M	ROR	EA,CL	WORD ROTATE EA RIGHT (CL) BITS
D3 11010011	MOD 010 R/M	RCL	EA,CL	WORD ROTATE EA LEFT THRU CARRY (CL) BITS
D3 11010011	MOD 011 R/M	RCR	EA,CL	WORD ROTATE EA RIGHT THRU CARRY (CL) BITS
D3 11010011	MOD 100 R/M	SHL	EA,CL	WORD SHIFT EA LEFT (CL) BITS
D3 11010011	MOD 101 R/M	SHR	EA,CL	WORD SHIFT EA RIGHT (CL) BITS
D3 11010011	MOD 110 R/M	(not used)		
D3 11010011	MOD 111 R/M	SAR	EA,CL	WORD SHIFT SIGNED EA RIGHT (CL) BITS
D4 11010100	00001010	AAM		ASCII ADJUST FOR MULTIPLY
D5 11010101	00001010	ADD		ASCII ADJUST FOR DIVIDE
D6 11010110		(not used)		
D7 11010111		XLAT	TABLE	TRANSLATE USING (BX)
D8 11011---	MOD --- R/M	ESC	EA	ESCAPE TO EXTERNAL DEVICE
E0 11100000		LOOPNZ/LOOPNE	DISP8	LOOP (CX) TIMES WHILE NOT ZERO/NOT EQUAL
E1 11100001		LOOPZ/LOOPE	DISP8	LOOP (CX) TIMES WHILE ZERO/EQUAL
E2 11100010		LOOP	DISP8	LOOP (CX) TIMES
E3 11100011		JCXZ	DISP8	JUMP ON (CX)=0
E4 11100100		IN	AL,PORT	BYTE INPUT FROM PORT TO REG AL
E5 11100101		IN	AX,PORT	WORD INPUT FROM PORT TO REG AX
E6 11100110		OUT	PORT,AL	BYTE OUTPUT (AL) TO PORT
E7 11100111		OUT	PORT,AX	WORD OUTPUT (AX) TO PORT
E8 11101000		CALL	DISP16	DIRECT INTRA SEGMENT CALL
E9 11101001		JMP	DISP16	DIRECT INTRA SEGMENT JUMP
EA11101010		JMP	DISP16,SEG16	DIRECT INTER SEGMENT JUMP
EB11101010		JMP	DISP8	DIRECT INTRA SEGMENT JUMP
EC11101010		IN	AL,DX	BYTE INPUT FROM PORT (DX) TO REG AL
ED11101010		IN	AX,DX	WORD INPUT FROM PORT (DX) TO REG AX
EE11101010		OUT	DX,AL	BYTE OUTPUT (AL) TO PORT (DX)
EF11101010		OUT	DX,AX	WORD OUTPUT (AX) TO PORT (DX)
F0 11110000		LOCK		BUS LOCK PREFIX
F1 11110001		(not used)		
F2 11110010		REPNZ		REPEAT WHILE (CX)≠0 AND (ZF)=0
F3 11110011		REPN		REPEAT WHILE (CX)≠0 AND (ZF)=1
F4 11110100		HLT		HALT
F5 11110101		CMC		COMPLEMENT CARRY FLAG
F6 11110110	MOD 000 R/M	TEST	EA,DATA8	BYTE TEST (EA) WITH DATA
F6 11110110	MOD 001 R/M	(not used)		
F6 11110110	MOD 010 R/M	NOT	EA	BYTE INVERT EA
F6 11110110	MOD 011 R/M	NEG	EA	BYTE NEGATE EA
F6 11110110	MOD 100 R/M	MUL	EA	BYTE MULTIPLY BY (EA), UNSIGNED
F6 11110110	MOD 101 R/M	IMUL	EA	BYTE MULTIPLY BY (EA), SIGNED
F6 11110110	MOD 110 R/M	DIV	EA	BYTE DIVIDE BY (EA), UNSIGNED
F6 11110110	MOD 111 R/M	IDIV	EA	BYTE DIVIDE BY (EA), SIGNED
F7 11110111	MOD 000 R/M	TEST	EA,DATA16	WORD TEST (EA) WITH DATA
F7 11110111	MOD 001 R/M	(not used)		
F7 11110111	MOD 010 R/M	NOT	EA	WORD INVERT EA
F7 11110111	MOD 011 R/M	NEG	EA	WORD NEGATE EA
F7 11110111	MOD 100 R/M	MUL	EA	WORD MULTIPLY BY (EA), UNSIGNED
F7 11110111	MOD 101 R/M	IMUL	EA	WORD MULTIPLY BY (EA), SIGNED
F7 11110111	MOD 110 R/M	DIV	EA	WORD DIVIDE BY (EA), UNSIGNED
F7 11110111	MOD 111 R/M	IDIV	EA	WORD DIVIDE BY (EA), SIGNED
F8 11111000		CLC		CLEAR CARRY FLAG
F9 11111001		STC		SET CARRY FLAG
FA 11111010		CLI		CLEAR INTERRUPT FLAG
FB 11111011		STI		SET INTERRUPT FLAG
FC 11111100		CLD		CLEAR DIRECTION FLAG
FD 11111101		STD		SET DIRECTION FLAG
FE 11111110	MOD 000 R/M	INC	EA	BYTE INCREMENT EA
FE 11111110	MOD 001 R/M	DEC	EA	BYTE DECREMENT EA
FE 11111110	MOD 010 R/M	(not used)		
FE 11111110	MOD 011 R/M	(not used)		
FE 11111110	MOD 100 R/M	(not used)		
FE 11111110	MOD 101 R/M	(not used)		

FE 11111110	MOD 110 R/M	(not used)		
FE 11111110	MOD 111 R/M	(not used)		
FF 11111111	MOD 000 R/M	INC	EA	WORD INCREMENT EA
FF 11111111	MOD 001 R/M	DEC	EA	WORD DECREMENT EA
FF 11111111	MOD 010 R/M	CALL	EA	INDIRECT INTRA SEGMENT CALL
FF 11111111	MOD 011 R/M	CALL	EA	INDIRECT INTER SEGMENT CALL
FF 11111111	MOD 100 R/M	JMP	EA	INDIRECT INTRA SEGMENT JUMP
FF 11111111	MOD 101 R/M	JMP	EA	INDIRECT INTER SEGMENT JUMP
FF 11111111	MOD 110 R/M	PUSH	EA	PUSH (EA) ON STACK
FF 11111111	MOD 111 R/M	(not used)		

REG IS ASSIGNED ACCORDING TO THE FOLLOWING TABLE:

16-BIT (W=1)	8-BIT (W=0)	SEGMENT REG
000 AX	000 AL	00 ES
001 CX	001 CL	01 CS
010 DX	010 DL	10 SS
011 BX	011 BL	11 DS
100 SP	100 AH	
101 BP	101 CH	
110 SI	110 DH	
111 DI	111 BH	

EA IS COMPUTED AS FOLLOWS: (DISP8 SIGN EXTENDED TO 16 BITS)

00	000	(BX) + (SI)	DS
00	001	(BX) + (DI)	DS
00	010	(BP) + (SI)	SS
00	011	(BP) + (DI)	SS
00	100	(SI)	DS
00	101	(DI)	DS
00	110	DISP16 (DIRECT ADDRESS)	DS
00	111	(BX)	DS
01	000	(BX) + (SI) + DISP8	DS
01	001	(BX) + (DI) + DISP8	DS
01	010	(BP) + (SI) + DISP8	SS
01	011	(BP) + (DI) + DISP8	SS
01	100	(SI) + DISP8	DS
01	101	(DI) + DISP8	DS
01	110	(BP) + DISP8	SS
01	111	(BX) + DISP8	DS
10	000	(BX) + (SI) + DISP16	DS
10	001	(BX) + (DI) + DISP16	DS
10	010	(BP) + (SI) + DISP16	SS
10	011	(BP) + (DI) + DISP16	SS
10	100	(SI) + DISP16	DS
10	101	(DI) + DISP16	DS
10	110	(BP) + DISP16	SS
10	111	(BX) + DISP16	DS
11	000	REG AX / AL	
11	001	REG CX / CL	
11	010	REG DX / DL	
11	011	REG BX / BL	
11	100	REG SP / AH	
11	101	REG BP / CH	
11	110	REG SI / DH	
11	111	REG DI / BH	

FLAGS REGISTER CONTAINS:

X:X:X:X:(OF):(DF):(IF):(TF):(SF):(ZF):X:(AF):X:(PF):X:(CF)


郑州