第 2 章　8086 系统结构

本章导读

- 8086 CPU 的结构
- 8086 系统的结构
- 8086 系统的配置
- 8086 CPU 的内部时序

8086 CPU 曾是使用广泛的 16 位微处理器。80386、80486 和 Pentium 系列、Core 系列都是从 8086 发展而来的，称为 80x86 系列。8086 是由 Intel 公司设计生产的，具有 40 个引脚的双列直插式封装芯片，内外数据总线都为 16 位，地址总线为 20 位，直接寻址为 1 MB。最早用于 IBM PC 中的 8088 CPU，其内部结构与 8086 基本相同，只是 8088 只有 8 条外部数据总线，因此也称为准 16 位微处理器。

本章主要介绍 8086 微处理机，详细讲述 8086 CPU 的结构，然后围绕以 8086 CPU 组成的计算机系统，介绍有关功能部件及其相互作用、外部引脚和系统配置，为了便于读者进一步了解 8086 执行指令的过程，本章还将介绍 8086 CPU 的内部时序。

2.1　8086 CPU 结构

2.1.1　8086 CPU 的内部结构

8086 CPU 由两部分组成，如图 2-1 所示，即指令执行部件（Execution Unit，EU）和总线接口部件（Bus Interface Unit，BIU）组成，在图中用点画线隔开。

指令执行部件主要由算术逻辑运算单元（ALU）、标志寄存器（FR）、通用寄存器组和 EU 控制电路 4 个部件组成，其主要功能是执行指令。

总线接口部件（BIU）主要由地址加法器、专用寄存器组、指令队列和总线控制电路 4 个部件组成，其主要功能是形成访问存储器的物理地址、访问存储器并取指令暂存到指令队列中等待执行，访问存储器或 I/O 端口读取操作数来参加 EU 运算或存放运算结果等。

传统的 CPU 在执行一个程序时，总是先从存储器中取出下一条指令，读出一个操作数（如指令需要操作数的话），然后执行指令，如图 2-2 所示。

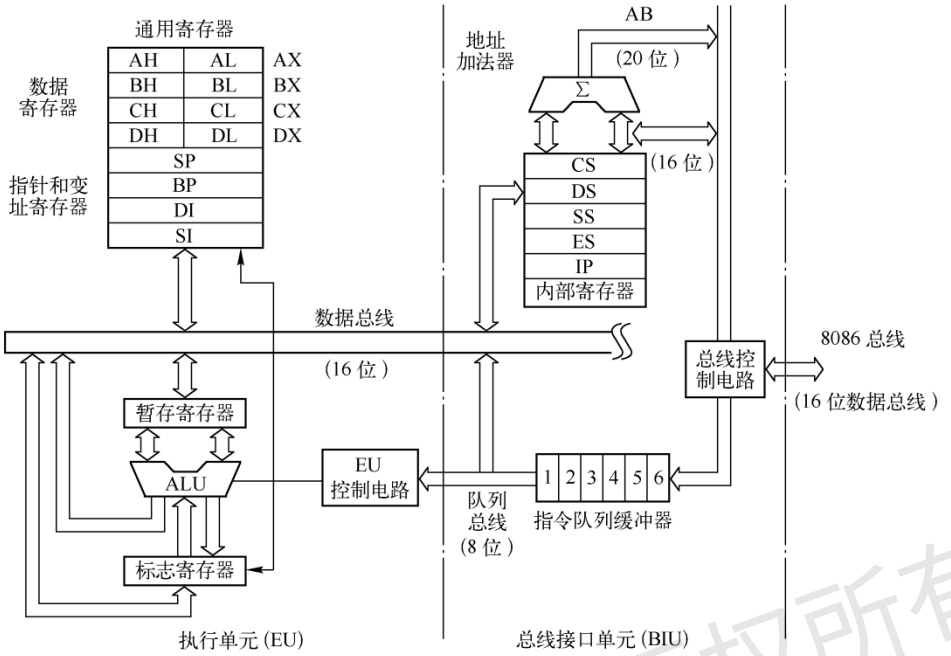

图 2-1　8086 CPU 的内部结构

在 8086 CPU 中，这些步骤分配给两个独立的处理单元进行，指令执行部件（EU）负责执行指令，总线接口部件（BIU）负责取指令、读出操作数和写入结果。这两个单元能够相互独立地工作，并在大多数情况下，大部分取指令和执行指令重叠进行，即在取指令的同时，指令执行部件也同时工作，这就有效地加快了系统的运算速率，如图 2-3 所示。

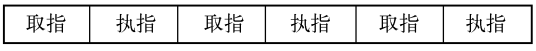

图 2-2　传统 CPU 的工作方式

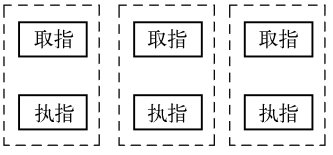

图 2-3　8086 CPU 工作方式

换句话说，执行部件（EU）在执行指令时不必通过访问存储器去取指令，而是从指令队列中取得指令代码，并分析执行它。若在指令执行过程中需要访问存储器或 I/O 端口，则 EU 只需向 BIU 送出访问存储器的逻辑地址。BIU 将根据 EU 要求形成访问存储器的物理地址，然后访问存储器或 I/O 端口，取得操作数后送到 EU 中参加运算，必要时可将运算结果再写回存储器中。所以，EU 实际上不与外界打交道，所有与外部的操作都在 BIU 控制下完成。

1．指令执行部件

EU 只负责执行指令。在一般情况下，指令是顺序执行的，EU 可源源不断地从指令队列中取得待执行的指令，达到满负荷地连续执行指令，从而节省“取指”时间。如果在指令执行过程中需要访问存储器取操作数，那么 EU 将访问地址送给 BIU 后，等待操作数到来后才能继续操作；遇到转移指令，BIU 会将指令队列中的后继指令作废。这时，EU 等待 BIU 重新从存储器中取出目标地址中的指令代码进入指令队列后，才继续执行指令。在这种情况下，EU 和 BIU 的并行操作会受到一定影响，这是采用重叠操作方式不可避免的现象。只要转移指令出现

率不是很高，两者的重叠操作仍然会取得良好效果。顺便指出，EU 处理的所有地址都是 16 位的，但是 BIU 能实现地址浮动，使 EU 可以访问 1 MB 的存储空间。

EU 的 ALU（算术逻辑运算单元）完成 8 位或 16 位的二进制数运算，运算结果通过内部总线送到通用寄存器组或 BIU 的内部寄存器，等待写入存储器。暂存寄存器用来暂存参加运算的操作数，经 ALU 运算后的结果状态如进位、溢出等信息置入 FR（标志寄存器）保存。

EU 控制电路负责从 BIU 的指令队列中取指令，并对指令译码，根据指令要求向 EU 内部各部件发出控制命令，以完成各条指令的功能。

2．总线接口部件

BIU 负责与外部存储器或 I/O 端口打交道。在正常情况下，BIU 通过地址加法器形成某条指令在存储器中的物理地址后，启动存储器，从给定的地址中取出指令代码，送 BIU 中的指令队列中等待执行，一旦指令队列中空出 2 字节，BIU 将自动进入读指令的操作，以填满指令队列。只要收到 EU 送来的操作数地址，BIU 就立即形成操作数的物理地址，完成读/写操作数或运算结果等功能。当遇到转移类指令执行转移时，BIU 将指令队列中尚存的指令“作废”，重新从存储器新的目标地址中取指令，并送指令队列中。

BIU 中的指令队列可以存放 6 字节的指令代码，一般应保证指令队列中总是填满指令，使得 EU 可以不断地得到等待执行的指令。20 位地址加法器专门用来完成由逻辑地址变换成物理地址的功能，实际上是进行一次地址加法，将两个 16 位的逻辑地址变换为 20 位的物理地址，从而使可寻址的存储空间达到 1 MB。

总线控制电路将 8086 CPU 的内部总线与外部总线相连，是 8086 CPU 与外部交换数据的必经之路，实际上包括 16 条数据总线、20 条地址总线和若干控制总线。CPU 正是通过这些总线与外部世界取得联系，从而形成各种规模的 8086 微型计算机。

关于 EU 和 BIU 中的寄存器结构，本书将在后续章节中专门讨论。

2.1.2　8086 CPU 的寄存器结构

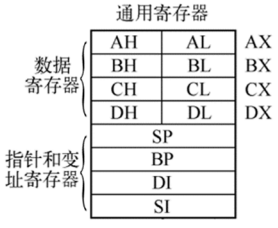

图 2-4　8086 通用寄存器

1．通用寄存器

微处理器的结构中许多寄存器，其作用是让程序暂存数据和地址。8086 CPU 的 EU 中有 8 个 16 位通用寄存器，分成两组，如图 2-4 所示，一组为数据寄存器，另一组为指针和变址寄存器。

数据寄存器由 AX、BX、CX 和 DX 构成，称为通用数据寄存器。这些通用数据寄存器除了具有保存数据作用，还各有特殊用途。通用数据寄存器既可以作为 16 位寄存器，也可以作为 8 位寄存器来使用，即把每个 16 位的通用寄存器分成高 8 位和低 8 位。低 8 位寄存器被命名为 AL、BL、CL 和 DL，高 8 位寄存器被命名为 AH、BH、CH 和 DH。寄存器一般存放 8 位数据，这样 8086 CPU 内部就有了 8 个 8 位寄存器。

① AX（Accumulator Register，累加器）：用来存放参加运算的数据和结果，在乘、除法运算、I/O 操作、BCD 数运算中有不可替代的作用。

【例 2-1】 AX 特殊用法举例。指令

```
MUL    BL
```

是一个 8 位的乘法指令，其功能为寄存器 AL×BL。其中，一个乘数一定放在 AL 中，另一个乘数可以放在 BL 中，也可以放在 BH、CL 等 8 位寄存器中，乘积一定放在 AX 中。

由于 AL 在某些 8 位指令中有不可替代的作用，具有 8 位 CPU 中的累加器功能，所以 AL 也被称为 8086CPU 中的 8 位累加器。

【例 2-2】 AX 特殊用法举例。指令

```
MUL   BX
```

是一个 16 位的乘法指令，其功能为寄存器 AX×BX。其中，一个乘数一定放在 AX 中，另一个乘数可以放在 BX 中，也可以放在 BX、CX 等 16 位通用寄存器中，乘积一定放在 DXAX 中，DX 中保存高 16 位。

② BX（Base Register，基址寄存器）：除了可作为数据寄存器，还可存放内存的逻辑偏移地址，而 AX、CX、DX 不能。

【例 2-3】 BX 特殊用法举例。指令

```
MOV   AL, [BX]
```

中，BX 所存内容为内存地址。如图 2-5 所示，如果 BX 的内容为 1000H，那么上述指令以 BX 的内容作为地址，从地址 1000H 的内存单元取数据给 AL。

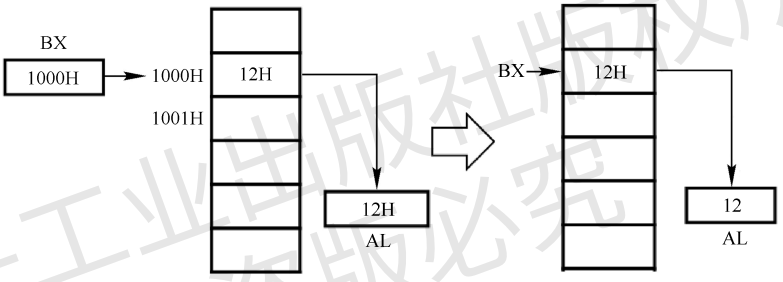

图 2-5　指令 MOV　AL, [BX]功能

说明：BX 作为地址时，这个地址是存储器单元的偏移地址。

BX 的内容可以作为地址指针的功能，是其他数据寄存器 AX、CX、DX 所不具备的。

③ CX（Counter Register，数据寄存器）：既可作为数据寄存器，又可在串指令和移位指令中作为计数用。

先考虑 C 语言编程中常常用到了循环控制语句：

```
for (i = 100; i > 0; i--)
{
    循环体程序段;
}
```

其中，计数器变量 i 初值为 100，每执行一次循环体，计数器变量 i 就会自动减 1，当循环体重复执行 100 次后，i 的内容递减为 0，循环过程结束。这种操作在 8086 体系中被称为串操作。

【例 2-4】 CX 特殊用法举例。指令

```
LOOP
```

其功能与上面的循环语句功能一致，其计数器 i 在 LOOP 语句中指定寄存器 CX，循环体结构如下：

```
again:
```

```
        循环体程序段;
        LOOP   again
```

其中，again 为标号，实际上是指令的符号位置，用来指定某一个指令的位置，这里指定的就是循环体程序段的第一条指令地址。

上面的程序段表示，执行完循环体程序段后就执行 LOOP 指令。LOOP 指令先对计数器 CX 做减 1 操作，并将结果送回 CX，再判断 CX 中的内容，若 CX 减 1 不为 0，则转到标号 again 处重复执行循环体程序段；若 CX 减 1 后为 0，则运行 LOOP 指令下面的程序段，这时循环过程结束。

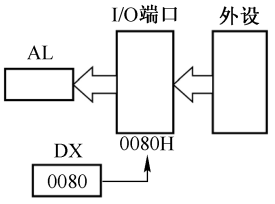

图 2-6　指令 IN　AL, DX 功能

④ DX（Data Register，数据寄存器）：除可作为通用数据寄存器外，还在乘、除法运算、带符号数的扩展指令中有特殊用途。

【例 2-5】 DX 特殊用法举例。指令

```
        IN   AL, DX
```

DX 所存内容可以作为 I/O 的端口地址。如图 2-6 所示，若 DX 的内容为 0080H，则上述指令就是从编号为 0080H 的 I/O 端口输入一个 8 位数据给 AL。这里，DX 的内容为 I/O 端口地址的功能，是其他寄存器如 AX、BX、CX 所不具备的。

数据寄存器 AX、BX、CX 和 DX 中，只有 BX 可以作为地址指针。考虑到实际编程中需要大量的地址指针，特别是涉及多个数组处理编程，只有 BX 指示或寻址数组中的元素是远远不够的，为此 8086 CPU 准备了另一组 16 位寄存器，称为指针和变址寄存器，由 SI、DI、BP 和 SP 组成，其特殊功能是存放存储器地址。

⑤ SI（Source Index，源变址寄存器）：用于存放内存的逻辑偏移地址（隐含的逻辑段地址在数据段寄存器中），也可存放数据。

⑥ DI（Destination Index，目标变址寄存器）：用于存放内存的逻辑偏移地址（隐含的逻辑段地址在数据段寄存器中），也可存放数据。

⑦ BP（Base Pointer，基址指针）：用于存放内存的逻辑偏移地址（隐含的逻辑段地址在堆栈段寄存器中）。

⑧ SP（Stack Pointer，堆栈指针）：用于存放栈顶的逻辑偏移地址（隐含的逻辑段地址在堆栈段寄存器中）。

SI、DI 和 BP 与 BX 的功能差不多，但 SP 有明显差异，主要用于指示堆栈栈顶。这 4 个寄存器详细功能将在第 3 章讲述。

为了更好地管理存储器，8086 CPU 把对应的存储空间分成几个逻辑段，存放在上述指针或变址寄存器中的内容往往是在某逻辑段中寻址的偏移地址。例如，一条 ADD 指令可以在当前数据段中指定它的一个操作数，办法之一是把该操作数的偏移量放在一个指示器或变址寄存器中。当然，这些寄存器也可以存放 16 位数据。

对于一些指令，通用寄存器具有一致性。例如，ADD 指令可将任意两个 8 位或 16 位通用寄存器的内容相加，结果可存放到这两个寄存器中的任何一个中。为了缩短指令代码长度，8086 CPU 的少数指令把某些通用寄存器作为专用。例如，串操作指令总是用 CX 寄存器存放串的长度，并在串操作指令执行过程中，CX 寄存器专用于计数。这样，所有串操作指令不必再在指令中指定 CX 寄存器，因而缩短了串操作指令的代码长度。如果在指令中没有明显指出

但指令中需要使用这些寄存器，通常称为“隐含寻址”。

隐含寻址实际上是在某类指令中指定某些通用寄存器作为特殊用法。例如，8 位乘法指令 MUL　BL，指令中没有标出的寄存器为 AL 和 AX，但 AL 被指定为一个乘数，而 AX 作为乘积；16 位的乘法指令 MUL　BX，指令中没有标出的寄存器为 AX 和 DX，但指定 AX 作为一个乘数，DXAX 作为乘积。程序设计者在编制程序时需遵循这些规定，将某些特殊数据放在特定寄存器中，才能正确地执行这些指令。这样也许会给程序设计者带来一些麻烦，但因其采用“隐含”方式，能有效地缩短指令代码的长度，并有可能提高指令的运行速度。

在 8086 CPU 中，有些寄存器具有上述隐含性质，即相应的指令中不必给出寄存器名；另有一些寄存器虽也具有特殊用途，但不能隐含寻址，指令中使用这些寄存器时必须给出它们的寄存器名，如表 2-1 所示。

表 2-1　寄存器的特殊用途和隐含性质

寄存器名	特 殊 用 途	隐含性质
AX，AL	在输入/输出指令中作为数据寄存器	不能隐含
	在乘法指令中存放被乘数或乘积，在除法指令中存放被除数或商	隐含
AH	在 LAHF 指令中，作为目标寄存器	隐含
AL	在十进制数运算指令中作为累加器	隐含
	在 XLAT 指令中作为累加器	隐含
BX	在间接寻址中作为基址寄存器	不能隐含
	在 XLAT 指令中作为基址寄存器	隐含
CX	在串操作指令和 LOOP 指令中作为计数器	隐含
CL	在移位/循环移位指令中作为移位次数计数器	不能隐含
DX	在字乘法/除法指令中存放乘积高位或被除数高位或余数	隐含
	在间接寻址的输入/输出指令中作为地址寄存器	不能隐含
SI	在字符串运算指令中作为源变址寄存器	隐含
	在间接寻址中作为变址寄存器	不能隐含
DI	在字符串运算指令中作为目标变址寄存器	隐含
	在间接寻址中作为变址寄存器	不能隐含
BP	在间接寻址中作为基址指针	不能隐含
SP	在堆栈操作中作为堆栈指针	隐含

2．段寄存器

8086 CPU 总线接口部件（BIU）有如下 4 个 16 位段寄存器。

- CS（Code Segment，代码段寄存器）：存放程序代码段起始地址的高 16 位。
- DS（Data Segment，数据段寄存器）：存放数据段起始地址的高 16 位。
- SS（Stack Segment，堆栈段寄存器）：存放堆栈段起始地址的高 16 位。
- ES（Extended Segment，扩展段寄存器）：存放扩展数据段起始地址的高 16 位。

段寄存器中存的也是地址，专门用于定位一个段的内存区域，通过指示这个区域的首个单元地址来定位一个段。在 8086 体系中，一个段所代表的内存区域最大为 64K。

8086 具有 1 MB 的存储空间，但放在指令指示器和变址寄存器中的地址都只有 16 位。16 位地址不能直接提供 1 MB 存储器寻址，只能在一个特定的 64 KB 段的偏移量中寻址。由此，划分地址段并且确定偏移量寻址是在哪一段中进行的就变得至关重要。在 8086 系统中，1 MB 存储空间可被分成许多逻辑段，每段最长为 64 KB，这些逻辑段可在整个 1 MB 存储空间中浮

动。同时，划分的各逻辑段首地址的高 16 位存放在该段寄存器中，高 16 位地址又称为段基址。这样，代码段寄存器 CS 用来存放当前代码段的段基址，数据段寄存器 DS 用来存放当前数据段的段基址，扩展段寄存器 ES 用来存放扩展段的段基址，堆栈段寄存器 SS 用来存放堆栈段的段基址。

系统中只设 4 个段寄存器，任何时候 CPU 只能识别当前可寻址的 4 个逻辑段。通常，代码段用来存放可执行的指令，数据段和扩展段用来存放参加运算的操作数和运算结果，堆栈段作为程序执行中需要使用的堆栈，即在存储器中开辟的堆栈区。如果程序量或数据量很大，超过 64 KB，就需要定义多个代码段、数据段、扩展段和堆栈段，只是在 4 个段寄存器中存放的应该是当前正在使用的逻辑段的段基址，必要时可以修改这些段寄存器的内容，以扩大程序的规模，这样就可以访问 8086 系统的 1 MB 存储器。

3．标志寄存器

8086 CPU 设置了一个 16 位标志寄存器（FR），如图 2-7 所示，其中规定了 9 个标志位，用来存放运算结果特征和控制 CPU 操作。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
//	//	//	//	OF	DF	IF	TF	SF	ZF	//	AF	//	PF	//	CF

图 2-7　标志寄存器（FR）格式

标志寄存器（FR）中存放的 9 个标志位可以分成两类。一类为状态标志，用来表示运算结果的特征，它们是 CF、PF、AF、ZF、SF 和 OF；另一类为控制标志，用来控制 CPU 的操作，它们是 IF、DF 和 TF。

（1）CF（Carry Flag），进位标志位

CF=1，表示本次运算中最高位（D_{15} 或 D_7）有进位（加法运算时）或有借位（减法运算时）。CF 标志可以通过 STC 指令置位，通过 CLC 指令复位（清除进位标志），还可通过 CMC 指令将当前 CF 标志取反。

（2）PF（Parity Flag，奇偶校验标志位）

PF=1，表示本次运算结果中有偶数个“1”；PF=0，表示本次运算结果中有奇数个“1”。

（3）AF（Auxiliary Carry Flag，辅助进位标志位）

AF=1，表示运算结果的 8 位数据中，低 4 位向高 4 位有进位（加法运算时）或有借位（减法运算时），这个标志位只在十进制数即 BCD 码运算中有用。

（4）ZF（Zero Flag，零标志位）

ZF=1，表示本次运算结果为 0；ZF=0，运算结果非 0。

（5）SF（Sign Flag，符号标志位）

SF=1，表示本次运算结果的最高位（D_{15} 或 D_7）为“1”，否则 SF=0。

（6）OF（Overflow Flag，溢出标志位）

OF=1，表示本次算术运算结果产生溢出，否则 OF=0。所谓溢出，就是字节运算的结果超出了范围–128～+127，或者字运算超出了范围–32768～32767。在加法运算时，当次高位向最高位有进位而最高位没有向前进位时，便产生溢出，于是 OF=1；或者，当次高位向最高位无进位而最高位向前有进位时，同样产生溢出，于是 OF=1。在减法运算时，当判断出最高位需要借位而低位并不向最高位产生借位时，OF=1；或者，当判断出低位从最高位有借位而最高

位并不需要从更高位借位时，OF=1。计算机在进行算术运算时，判断是否溢出的一个简单方法是：运算结果是否合理。例如，当两个正数相加，和为负数时，OF=1，否则 OF=0。**说明：不管有符号数或无符号数，计算机在运算时会影响 OF 标志，但对编程者来说，该标志仅对带符号数有意义。**

① 算术操作和逻辑操作都会影响状态标志。为了对上述 6 个状态标志位有更具体了解，举两个例子。

【例 2-6】 加法运算一。

```
  0010 0011 0100 0101
+ 0011 0010 0001 1001
= 0101 0101 0101 1110
```

结果对各状态标志的影响如下：运算结果的最高位为 0，所以 SF=0；运算结果本身不为 0，所以 ZF=0；结果中所含 1 的个数为 9，即奇数个 1，所以 PF=0；最高位没有产生进位，所以 CF=0；D_3 位没有往 D_4 位产生进位，所以 AF=0；次高位没有往最高位产生进位，最高位也没有往前进位，所以 OF=0。

【例 2-7】 加法运算二。

```
  0101 0100 0011 1001
+ 0100 0101 0110 1010
= 1001 1001 1010 0011
```

结果对各状态标志的影响如下：运算结果的最高位为 1，所以 SF=1；运算结果本身不为 0，所以 ZF=0；结果中所含 1 的个数为 8，即偶数个 1，所以 PF=1；最高位没有产生进位，所以 CF=0；D_3 位向 D_4 位产生了进位，所以 AF=1；次高位往最高位产生了进位，而最高位没有往前产生进位，所以 OF=1。

【例 2-8】 逻辑“或”运算。

```
  0100 0100 0011 1001
OR 0100 0100 0011 1001
= 0100 0100 0011 1001
```

结果对各状态标志的影响如下：运算结果的最高位为 0，所以 SF=0；运算结果本身不为 0，所以 ZF=0；结果中所含 1 的个数为 5，即奇数个 1，所以 PF=0；因为是逻辑运算，不存在进位和溢出情况，所以 CF=0，AF=0，OF=0。

上面的例子说明：逻辑运算可以使标志位 CF 清零；如果想测试操作数中 1 的个数的奇偶性，可以采用两个相同内容的操作数做逻辑运算。

② 标志位 CF 可以直接参与运算。标志位 CF 不仅可以作为运算后的标志，说明运算的结果状态，也可以直接参与运算。

【例 2-9】 32 位数的加法。

为了书写方便，这里用十六进制列出算式。

```
  5B10 090A
+ 3A31 F706
= 9542 0610
```

8086 CPU 指令系统中只有 16 位加法指令，为了完成 32 位加法，必须采用编程的手段，

通过两次 16 位加法运算来实现 32 位加法。具体做法是：先对两个加数的低 16 位相加，得出和的低 16 位；再对两个加数的高 16 位相加，得出和的高 16 位。但是上面的运算中存在低 16 位向高 16 位的进位，这个进位不能被忽视。为此，8086 CPU 指令系统中配置了两种加法指令：一种是 ADD 指令，只能完成一般 16 位加法；另一种是 ADC 指令，把进位标志及两个数同时加在一起。因此在编程时，先采用 ADD 实现低 16 位的相加，再采用 ADC 完成高 16 位数相加，这样就完成了 32 位数的相加。

可见，标志位 CF 起到扩充多位数的算术加或减的作用。

③ 通过标志位比较数的大小。比较两个数的大小是编程中常常遇到的情况，参与比较的数也存在“有符号数”和“无符号数”的区别。

比较两个无符号数的大小相对容易。先对两个操作数 x 和 y 做减法运算，即 $x-y$，再通过 ZF 标志位和 CF 标志位完成比较。过程如下：先判别 ZF 是否有效，若 ZF 有效（即 ZF=1），则 $x=y$，否则 $x\neq y$；若 ZF 无效，则判别 CF 标志位，若 CF 标志位为 1，则 $x<y$，反之 $x>y$。

对于有符号数，由于存在溢出情况，比较两个数的大小就复杂些了。先对操作数 x 与 y 做减法运算，再通过标志位 ZF 标志位判断两数是否相等。

若 ZF 有效（即 ZF=1），则 x 与 y 相等，否则（即 ZF=0）x 与 y 不相等。若 ZF 无效，则判断符号标志位 SF 和溢出标志位 OF。若没有溢出，OF=0，同时若 SF=0，两数相减的结果是正，则 $x>y$；SF=1，则两数相减的结果是负数，$x<y$。若有溢出，则 OF=1。由于溢出就是实际运算中的符号位丢失，因此当 SF=1 时，两数相减的结果为正，则 $x>y$，否则两数相减的结果就是负数，则 $x<y$。

表 2-2　有符号数比较大小

OF	SF	结果
0	0	$x>y$
0	1	$x<y$
1	0	$x<y$
1	1	$x>y$

综上所述，我们可以得出表 2-2。当 OF 和 SF 同类符号时，结果是 $x>y$，而当 OF 和 SF 异号时，结果是 $x<y$。因此，可以用下面的表达式表示有符号数比较大小的结果：若 OF⊕SF=0，则 $x>y$；若 OF⊕SF=1，则 $x<y$。符号“⊕”表示逻辑异或。

下面继续介绍其他 3 个控制标志。

（7）IF（Interrupt Flag，中断标志位）

IF=1，表示允许 CPU 响应可屏蔽中断。IF 标志可以通过 STI 指令置位，也可通过 CLI 指令复位。

（8）DF（Direction Flag，方向标志位）

在串操作指令中，DF=0，表示串操作指令地址指针自动增量，即串操作的地址由低地址向高地址进行；DF=1，表示地址指针自动减量，即串操作的地址是由高地址向低地址进行的。DF 标志位可以通过 STD 指令置位，也可通过 CLD 指令复位。

（9）TF（Trap Flag，单步标志位）

TF=1，表示控制 CPU 进入单步工作方式。在这种工作方式下，CPU 每执行完一条指令就自动产生一次内部中断，这在程序调试过程中非常有用。对 TF 标志的设定和清除没有专用指令，但可用编程间接达到目的。例如，先将标志寄存器的内容推入堆栈，设法对标志寄存器的某一位（即 TF 位）进行置 1 或清零操作，再利用堆栈指令将修改后的数据送入标志寄存器中，这样就可以对该标志进行置位和清零。

4．指令指针寄存器

16 位指令指针寄存器（IP）与传统的 8 位微处理器中的程序计数器（PC）相似。IP 中的

内容可由 BIU 自动修改，使之始终存有相对于当前代码段起点偏移量的下一条指令，即 IP 总是指向下一条待执行的指令。当然，由于指令队列的缘故，这个定义并不十分确切。在正常执行过程中，IP 中存有 BIU 要取出的下一条指令的偏移量。程序是不能直接访问 IP 的，但可通过某些指令修改 IP 的内容，这类指令主要是程序控制类指令。该类指令可将转移目标地址送入 IP 中，以实现程序的转移，也可以将 IP 内容压入堆栈或从堆栈中弹出。

2.1.3 8086 CPU 的引脚及功能

8086 CPU 是 16 位 CPU，采用高性能的 N 沟道、耗尽型负载的硅栅工艺（HMOS）制造。由于受当时制造工艺的限制，部分引脚采用分时复用的方式，构成了 40 条引脚的双列直插式封装，如图 2-8 所示。

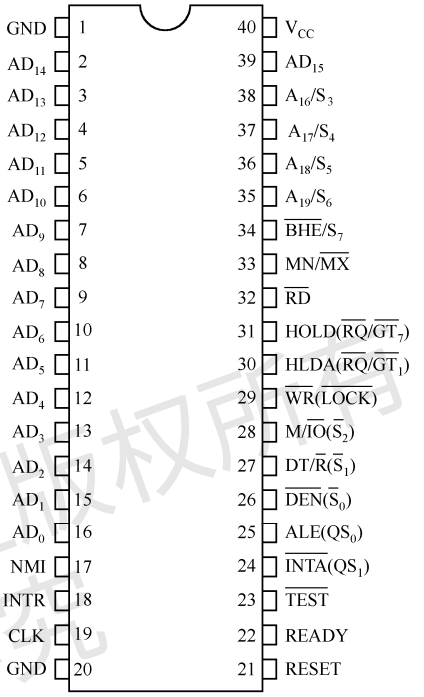

图 2-8　8086 CPU 引脚

8086 CPU 可以工作在最小模式和最大模式下，因此有 8 条引脚（24～31）在上述两种工作模式中具有不同的功能，图 2-8 括号中所示为最大模式下被重新定义的控制信号。

1．引脚功能

AD_{15} ~ AD_0（Address Data Bus）：分时复用的地址数据总线。传送地址时以三态输出，传送数据时可以以双向三态输入/输出。

A_{19} / S_6，A_{18} / S_5，A_{17} / S_4和A_{16} / S_3（Address/ Status）：分时复用的地址/状态线。作为地址线用时，A_{19} ~ A_{16} 与 AD_{15} ~ AD_0 一起构成访问存储器的 20 位物理地址。当 CPU 访问 I/O 端口时，A_{19} ~ A_{16} 保持为“0”（低电平）。作为状态线用时，S_6 ~ S_3 用来输出状态信息，其中 S_3 和 S_4 表示当前使用的段寄存器名，如表 2-3 所示，S_3S_4 =10 时表示当前正在使用 CS 寄存器对存储器寻址，或者是当前正在对 I/O 端口或中断矢量寻址。S_5 用来表示中断标志状态，当 IF=1 时，S_5 =1。S_6 恒保持为 0。

$\overline{BHE}$ / S_7（Bus High Enable/Status）：总线高位有效信号（三态输出，低电平有效），表示当前高 8 位数据总线上的数据有效。当读/写存储器或 I/O 端口以及中断响应时，$\overline{BHE}$ 与地址线 AD_0 配合表示当前总线使用情况，如表 2-4 所示。非数据传输期间，S_7 输出状态信息，低电平有效，在 CPU 处于保持响应期间被设置为高阻抗状态。

表 2-3　S_4、S_3 状态编码

S_4S_3	段寄存器
00	ES
01	SS
10	CS（I/O，INT）
11	DS

表 2-4　$\overline{BHE}$ 和 AD_0 编码的含义

$\overline{BHE}$	AD_0	总线使用情况
0	0	16 位数据总线上进行字节传送
0	1	高 8 位数据总线上进行字节传送
1	0	低 8 位数据总线上进行字节传送
1	1	无效

$\overline{RD}$（Read）：读信号（三态输出，低电平有效），表示当前 CPU 正在读存储器或 I/O 端口。

$\overline{WR}$（Write）：写信号（三态输出，低电平有效），表示当前 CPU 正在写存储器或 I/O 端口。

M / $\overline{IO}$（Memory/IO）：存储器或 I/O 端口访问信号（三态输出）。M / $\overline{IO}$ 为高电平时，表示当前 CPU 正在访问存储器；M / $\overline{IO}$ 为低电平时，表示当前 CPU 正在访问 I/O 端口。

READY：准备就绪信号（由外部输入，高电平有效），表示 CPU 访问的存储器或 I/O 端口已准备好传输数据。当 READY 无效时，要求 CPU 插入一个或多个等待周期 T_w，直到 READY 信号有效为止。

INTR（Interrupt Request）：中断请求信号（由外部输入，电平触发，高电平有效）。INTR 有效时，表示外部设备向 CPU 发出中断请求，CPU 在每条指令的最后一个时钟周期对 INTR 进行测试。一旦测试到有中断请求，并且当中断允许标志 IF=1 时，则暂停执行下条指令，转入中断响应周期。

$\overline{INTA}$（Interrupt Acknowledge）：中断响应信号（向外部输出，低电平有效），表示 CPU 响应了外部发来的 INTR 信号。在中断响应总线周期，它可作为读取中断矢量的选通信号。

NMI（Non-Maskable Interrupt Request）：不可屏蔽中断请求信号（由外部输入，边沿触发，正跳变有效）。NMI 不受中断允许标志的限制，CPU 一旦测试到 NMI 请求信号，待当前指令执行完，就自动从中断入口地址表中找到类型 2 中断服务程序的入口地址，并转去执行。NMI 是一种比 INTR 高级的中断请求。

$\overline{TEST}$：测试信号（由外部输入，低电平有效）。当 CPU 执行 WAIT 指令时（WAIT 指令使处理器与外部硬件同步），每隔 5 个时钟周期对 TEST 进行一次测试，若测试到该信号无效，则 CPU 继续执行 WAIT 指令，即处于空闲等待状态；若 CPU 测到 TEST 输入为低电平，则转而执行 WAIT 的下一条指令。由此可见，TEST 对 WAIT 指令起到监视的作用。

RESET：复位信号（由外部输入，高电平有效），至少要保持 4 个时钟周期。CPU 接收到该信号后，停止进行操作，并对寄存器 FR、IP、DS、SS、ES 及指令队列清零，而将 CS 设置为 FFFFH。当复位信号变为低电平时，CPU 从 FFFF0H 开始执行程序。由此可见，采用 8086 CPU 计算机系统的启动程序就保持在开始的存储器中。

ALE（Address Latch Enable）：地址锁存允许信号（向外部输出，高电平有效），在最小模式系统中作为地址锁存器 8282/8283 的片选信号。

DT / $\overline{R}$（Data Transmit/Receive）：数据发送/接收控制信号（三态输出），在最小模式系统中用来控制数据收发器 8286/8287 的数据传送方向。若 DT / $\overline{R}$ 为高电平，表示数据从 CPU 向外部输出，即完成写操作；若 DT / $\overline{R}$ 为低电平，表示数据从外部向 CPU 输入，即完成读操作。

$\overline{DEN}$（Data Enable）：数据允许信号（三态输出，低电平有效），在最小模式系统中作为数据收发器 8286/8287 的选通信号。

HOLD（Hold Request）：总线请求信号（由外部输入，高电平有效且向 CPU 请求使用总线）。

HLDA（Hold Acknowledge）：在最小模式系统中表示有其他共享总线的处理总线请求响应信号（向外部输出，高电平有效）。CPU 一旦测试到有 HOLD 请求时，就在当前总线周期结束时，使 HLDA 有效，表示响应这一总线请求，并且立即让出总线使用权，CPU 中的 EU 可以继续工作到下一次要求使用总线为止。CPU 只有当 HOLD 无效时，才将 HLDA 置成无效，并且收回对总线的使用权，继续自己的操作。

MN / $\overline{MX}$（Minimum/Maximum）：工作模式选择信号（由外部输入），为高电平时，表示 CPU 工作在最小模式系统中；为低电平时，表示 CPU 工作在最大模式系统中。

CLK（Clock）：主时钟信号（由时钟发生器 8284 输入）。8086 CPU 可以使用的时钟频率

随芯片型号不同而异，8086 为 5 MHz，8086-1 为 10 MHz，8086-2 为 8 MHz。

V_{CC}（电源）：8086 CPU 只需要单一的+5 V 电源，由 V_{CC} 输入。

2．8086 CPU 工作在最大模式系统时，有 8 个引脚（24～31）要重新定义

$\overline{S}_2$、$\overline{S}_1$、$\overline{S}_0$（Bus Cycle Status）：总线周期状态信号（三态输出），在最大模式系统中由 CPU 传送给总线控制器 8288，8288 对它们译码后代替 CPU 输出相应的控制信号。详细情况将在本章 2.2 节中讨论。

$\overline{LOCK}$：封锁信号（三态输出，低电平有效）。$\overline{LOCK}$ 有效时，表示 CPU 不允许其他总线主控者占用总线。这个信号由软件设置。当在指令前加上 LOCK 前缀时，则在执行这条指令期间 $\overline{LOCK}$ 保持有效，即在此指令执行期间，CPU 封锁其他主控者使用总线。

$\overline{RQ}$ / $\overline{GT}_0$ 和 $\overline{RQ}$ / $\overline{GT}_1$（Request/Grant）：请求/同意信号（双向，低电平有效）。该信号为输入时，表示其他主控者向 CPU 请求使用总线；为输出时，表示 CPU 对总线请求的响应信号。两条线可同时与两个主控者相连，内部保证 $\overline{RQ}$ / $\overline{GT}_0$ 比 $\overline{RQ}$ / $\overline{GT}_1$ 有较高优先级。

QS_1 和 QS_0（Instruction Queue Status）：指令队列状态（向外部输出），表示 CPU 中指令队列当前的状态，如表 2-5 所示。设置这两个引脚的目的是让外部的设备监视 CPU 内部的指令队列，可让协处理器 8087 进行指令的扩展处理。

表 2-5　QS_1 和 QS_0 编码的含义

QS_1	QS_0	编码含义
0	0	无操作
0	1	从队列中取第一字节
1	0	队列已空
1	1	从队列中取后续字节

2.2　8086 CPU 的结构和配置

8086 CPU 是一个微处理器而不是一台微型计算机。微处理器并不包含任何存储单元或输入/输出接口。做一个形象的比喻，微处理器能够思考、判断，但是如果没有存储器就不能记忆，没有输入/输出接口就不能听（输入）或说（输出）。本节将简要介绍 8086 的存储系统和输入/输出结构，结合 8086 基本总线接口器件，给出 8086 最小模式系统和最大模式系统配置。

2.2.1　8086 存储器结构

8086 CPU 的存储器是一个最多可寻址的 1 MB 存储空间，系统为每字节分配一个 20 位的物理地址（对应的十六进制数地址范围为 00000H～FFFFFH）。在存储器中，任何两个相邻的字节被定义为一个字。在一个字中，每字节都有一个地址，并且这两个地址中的较小的一个被用来作为该字的地址。由表 2-6 可以看出，一个字的起始地址可以从偶地址开始，如数 6B07H，也可以从奇地址开始，如数 3E60H。并且，较高存储器地址的字节存放该字的高 8 位，较低存储器地址的字节存放该字的低 8 位。

表 2-6　字在存储器中的例子

内存地址	存储器	说　明	内存地址	存储器	说　明
00000H	07	低字节，字以偶地址开始	00004H	—	—
00001H	6B	高字节，表示的数为	00005H	60	低字节，字以奇地址开始
00002H	—	—	00006H	3E	高字节，表示的数为
00003H	—	—	00007H	—	—

1．存储器的组成

在 8086 系统中，存储器采用分体结构，即 1 MB 的存储空间分成两个 512 KB 的存储体，一个存储体中包含偶数地址，另一个存储体包含奇数地址。两个存储体采用字节交叉编址方式，如表 2-7 所示。

表 2-7　两个存储体采用交叉编址方式

奇地址	D_{15}～D_8	D_7～D_0	偶地址
00001H	—	—	00002H
00003H	—	—	00003H
00005H	—	—	00006H
…	512K×8　奇地址存储体（A_0=1）	512K×8　偶地址存储体（A_0=1）	…
FFFFFH	—	—	FFFFEH

对于任何一个存储体，只要 19 位地址（A_{19} ~ A_1）就够了。地址 A_0 用来区分当前访问的是哪一个存储体：A_0=0，表示访问偶地址存储体；A_0=1，表示访问奇地址存储体。8086 CPU 允许访问存储器中的 1 字节，也允许访问存储器中的 1 个字（相邻两字节），要求同时访问两个存储体，各取出 1 字节的信息。这时只用 A_0 控制读/写操作就不够了，为此增设了总线高位有效控制信号 $\overline{BHE}$。当 $\overline{BHE}$ 有效时，选定奇地址存储体，存储体内地址由 A_{19} ~ A_1 确定。当 A_0=0 时，选定偶地址存储体，存储体内地址同样由 A_{19} ~ A_1 确定。注意，偶地址存储体固定与低 8 位数据总线（D_7 ~ D_0）相连，因此被称为低字节存储体；奇地址存储体固定与高 8 位数据总线（D_{15} ~ D_8）相连，因此被称为高字节存储体。$\overline{BHE}$ 与 A_0 互相配合（如表 2-8 所示），使 CPU 可访问一个存储体中的 1 字节或同时访问两个存储体中的 1 个字。

两个存储体与总线之间的连接如图 2-9 所示。显然，奇地址存储体的片选端 $\overline{SEL}$ 受控于 8086 CPU 的 $\overline{BHE}$，偶地址存储体的片选端 $\overline{SEL}$ 受控于地址线 A_0。

表 2-8　$\overline{BHE}$ 与 A_0 组合对应的控制

$\overline{BHE}$	A_0	对应的操作
0	0	从偶地址读/写 1 个字
0	1	从奇地址读/写 1 字节
1	0	从偶地址读/写 1 字节
0 1	1 0	从奇地址读/写 1 个字（分两次读/写）

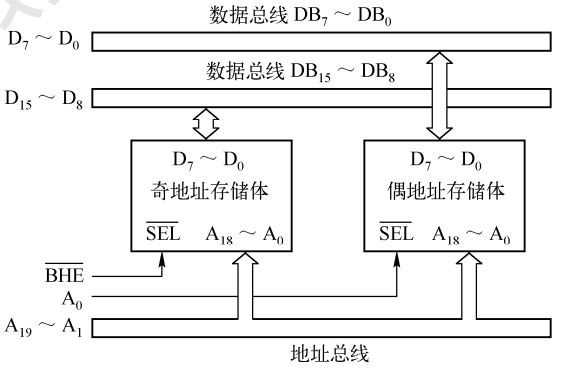

图 2-9　存储体与总线的连接

8086 的有些指令是访问（读或写）字节的，有些指令是访问字的。在同一个时间，8086 从存储器中取出的信息数量总是 16 位的，且该 16 位数据是在存储器中以偶地址开头的 2 字节的内容。当 8086 要访问字节时，在被读出的 16 位数据中，只要忽略高 8 位或低 8 位就可得到所要的 1 字节信息，如图 2-10(a)和(b)所示。当 8086 要访问 1 个字而这个字始于偶地址时，只要使 A_0=0，$\overline{BHE}$ = 0，就可一次访问到该字的内容，如图 2-10(c)所示；当要访问的字始于奇地址时，情况比较复杂，必须对两个连续的偶地址字做两次存储器访问，每次访问忽略不需要的 1 字节，并保留剩余的 1 字节，然后变换得到完整的一个字的信息，如图 2-10(d)所示。

必须指出，8086 的编程并不涉及这些细节，一条指令只是请求访问一个特定的字节或字，必须要做的操作都是在处理器控制下自动实现的。

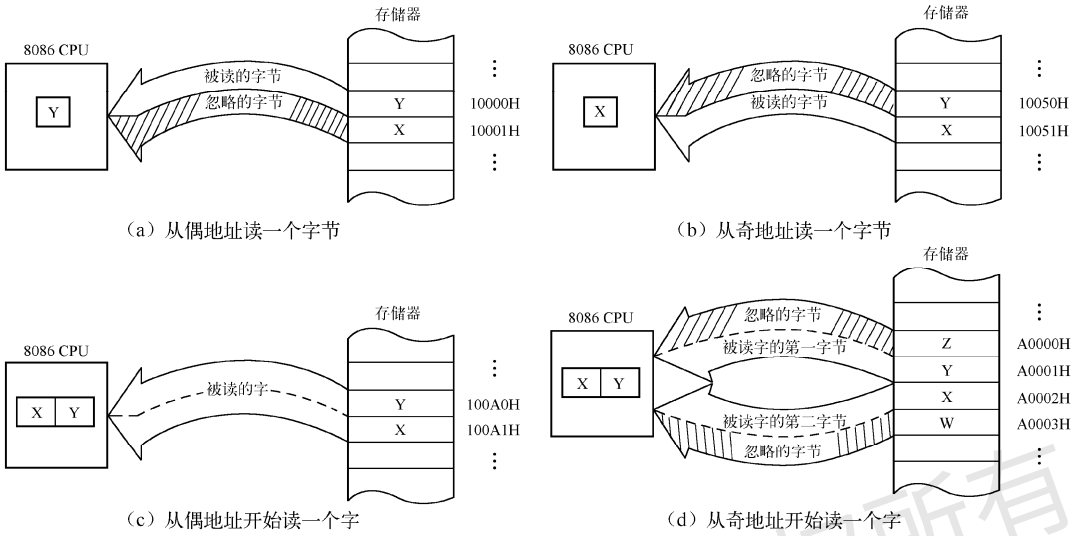

图 2-10　读存储器操作过程

如上所述，在字访问情况下，对奇地址存放的字需要进行两次读/写操作，对偶地址存放的字仅需要一次读/写操作。为了加快程序的运行速度，希望被访问的存储器的字地址为偶地址。通常，这种从偶地址开始的字称为“对准字”，而从奇地址开始的字称为“非对准字”。

2．存储器的分段

8086 的寻址空间是 1 MB，因此要对整个空间寻址需要 20 位长的地址码，但是所有寄存器都是 16 位寄存器。这样，如果仅是一个 16 位寄存器的内容寻址，就只能寻址 64 KB。要达到对 1 MB 空间的寻址，8086 采用分段并附以地址偏移量的办法形成 20 位的物理地址，得到对 1 MB 空间的寻址。被划分的存储器段称为逻辑段。这里先讨论段的特点。

① 在 8086 中，存储空间被设置成若干逻辑段，在一个编程任务中，存储空间一般被分为 4 个段，分别是代码段、附加段、堆栈段和数据段。实际编程应用时，根据需要，也可少于 4 个段。

② 段基地址。8086 中存在 8 位字节型变量、16 位字型变量和 32 位双字型变量，为了定位这些变量，这些变量在内存中的第一字节位置（或首地址）被作为其地址，称为段基址。同样一个 64 KB 的段，它在内存中第一字节的位置（或首地址）就是这个段的地址。

③ 为了管理段，8086 中设置了 4 个 16 位的段寄存器 CS、SS、DS、ES，用来指示段在内存中的位置。由于段基址是 20 位的，为了解决段寄存器是 16 位而段基址是 20 位的不兼容问题，8086 规定，段基址最低 4 位是 0000B，段寄存器只存段基址的第 4～19 位的信息，即存放相应段首地址的高 16 位（即段基址）。这样当段寄存器的内容确定后，在其内容最低位补 4 位二进制的 0000B，就可以生成段基址，相当于段寄存器内容乘以 16 或段寄存器内容左移 4 位。这样就可以确定该段的位置了。

④ 当段寄存器内容确定后，段的寻址范围已经确定，8086 规定其容量不大于 64 KB，同时通过修改段寄存器内容，可以使逻辑段在整个存储空间中浮动。各逻辑段之间可以紧密相

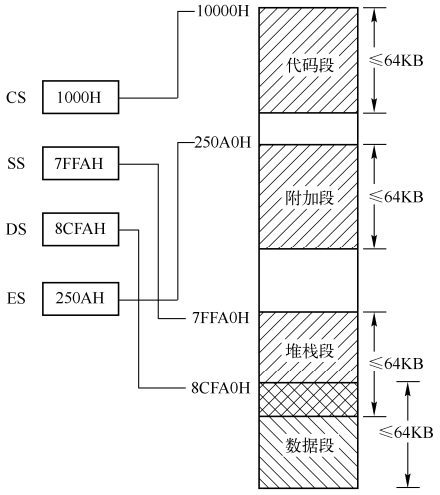

图 2-11　存储器分段的例子

连，可以中间有间隔，也可以相互重叠（部分重叠甚至完全重叠）。如图 2-11 所示，当前有效的代码段、附加段、堆栈段和数据段的段基址分别为 1000H、250AH、7FFAH 和 8CFAH。

在 8086 中，存储空间被设置成若干逻辑段，每个物理地址可被包含在一个逻辑段中，也可以包含在多个相互重叠的逻辑段中。

3．物理地址和逻辑地址

如何在存储器分段的前提下，使用 16 位的地址寄存器方便地在 1 MB 存储器空间中定位或找到目标存储单元呢？为了解决上面问题，我们先给出和存储器地址有关的两个概念。

① 物理地址。物理地址指存储器的绝对地址。在 8086 中，物理地址是 20 位的，直接对应 CPU 的地址线第 0～19 位，因此物理地址是 CPU 访问存储器的实际寻址地址，即存储单元对应 20 位的物理地址，其物理地址分布为 00000H～FFFFFH。

② 逻辑地址。图 2-12 清晰给出了物理地址到逻辑地址的演化过程。

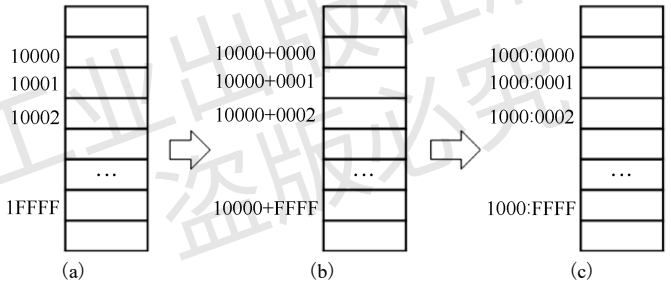

图 2-12　物理地址到逻辑地址的演化过程

图 2-12(a)是内存中的某一段，物理地址为 10000H～1FFFFH，段的首地址为 10000H。由于 8086 对段的规定，低 4 位默认为 0H（十六进制），高 16 位的 1000H 为段基址。段基址通常存在段寄存器中。其实，规定段首地址最低 4 为 0 的重要原因是段寄存器是 16 位的。而从段寄存器内容通过低位补 0 就可以直接得到段首地址。

图 2-12(b)是图 2-12(a)的另一种表示，把每个内存单元地址用段首地址和一个 16 位的地址之和来表示，这 16 位地址就是偏移地址。偏移地址即目标地址和段首地址之间的距离，或目标地址和段首地址之间的差值。比如，目标地址是 10002H 的内存单元，其偏移地址为 10002H−10000H =0002H。**注意：**由于目标地址是一个段的段内单元地址，因此其偏移地址最大不可能超过 16 位，这样段内的偏移地址分布就是 0000H～FFFFH。

图 2-12(c)是图 2-12(b)的另一种表示法。既然段首地址最低 4 位为 0，没有其他什么信息量，因此段首地址可以用 16 位的段基址表示，图 2-12(b)中的加号用冒号代替，因此图 2-12(c)所示段的第 2 号单元的地址表示为 1000H:0002H，显然它与该单元的物理地址 10002H 之间的关系是 1000H×10H+0002H。这里的 10H 为十进制数 16，表示 1000H 低位补 4 个二进制 0。

上面的 1000H:0002H 就是逻辑地址。因此，逻辑地址有两个分量：段基址和偏移地址，逻辑地址的格式是“段基址:偏移地址”。如图 2-13 所示，物理地址和逻辑地址之间的关系是：

物理地址=段基址×16+偏移地址

乘以 16 相当于 16 位的段基址最低位后添 4 个“0”。

采用分段结构的存储器中，任何一个逻辑地址都由段基址和偏移地址两部分构成，都是无符号的 16 位二进制数，程序设计时采用逻辑地址。8086 在程序运行时，根据操作属性，会自动确定目标单元的段寄存器。

如图 2-14 所示，在 CPU 运行过程中，当取指令时，选择代码段寄存器 CS，再与指令指针 IP 的内容一起形成指令所在单元的 20 位物理地址；当进行堆栈操作时，CPU 选择堆栈段寄存器 SS，再与堆栈指针 SP 或者基址指针 BP 一起形成 20 位堆栈指针；当往内存写一个数据或者从内存读一个数据时，CPU 会选择数据段寄存器 DS，再与变址寄存器 SI 和 DI 或者通用寄存器 BX 中（或指令中）的偏移值一起构成操作数所在存储单元的 20 位物理地址。

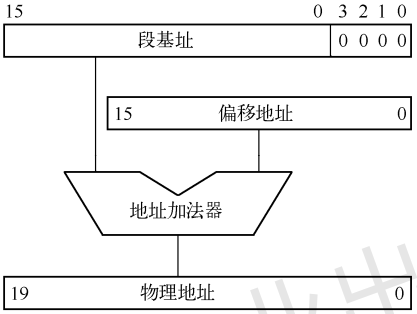

图 2-13　8086 CPU 地址完成过程

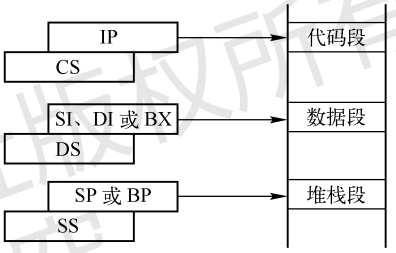

图 2-14　寄存器组合指向存储单元

8086 在执行复位信号后，从地址为 FFFF0H 开始执行程序。这个地址实际上是 CS 的内容（复位时置为 FFFFH）左移 4 位再加上 IP 的内容（复位时置为 0H）形成的。由于上述原因，在存储器安排时，将高地址端分配给 ROM（固化的只能读的存储器），在 FFFF0H 开始的几个单元中存放一条无条件转移指令，在复位时，自动转到系统初始化程序中。

4．堆栈段的使用

堆栈是在存储器中开辟的一个区域，用来存放需要暂时保存的数据和地址信息，采用“先进后出”或“后进先出”方式。8086 中的堆栈段是由段定义语句在存储器中定义的一个段，与其他逻辑段一样，可以在存储器的 1 MB 空间内任意浮动，堆栈段容量小于或等于 64 KB。段基址由堆栈寄存器 SS 指定，栈顶由堆栈指针 SP 指定。堆栈的地址增长方式是向下增长的，即随着堆栈内容的增加，堆栈指针的值是减小的。栈底设在存储器的高地址区，堆栈地址由高向低增长。

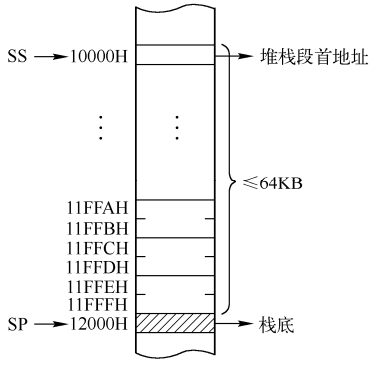

图 2-15　堆栈在存储器中的分布情况

若(SS)=1000H，(SP)=2000H（括号表示给定寄存器中存储的内容），则堆栈在存储器的分布情况如图 2-15 所示。

在 8086 中，堆栈仅以字为单位进行操作，并且堆栈中的数据项必须对准字存储，即低字

节在偶地址，高字节在奇地址，以保证每访问一次堆栈就能压入或弹出一个字的信息。

堆栈的主要操作是入栈操作（PUSH）和出栈操作（POP）。入栈操作时，总是先修改指针（SP－2⇒SP），再将信息入栈；出栈操作时，总是先将信息出栈，再修改指针（SP＋2⇒SP）。

2.2.2 8086 CPU 的输入/输出结构

8086 CPU 与外部设备的输入/输出是通过接口完成的。设计 CPU 时，人们并没有设计它与外部设备之间的接口部件，而是将输入/输出（I/O）端口设计成完全独立的部件，使它们成为通用的接口芯片。它们可将各种类型的输入或输出设备与 CPU 连接起来，构成完整的微机系统。专用的 I/O 端口芯片将在后面章节详细介绍，本节介绍 8086 CPU 的输入/输出结构。

8086 CPU 有一套灵活的 I/O 功能，不仅提供与存储空间分开的 I/O 空间，还提供与位于 I/O 空间内的设备进行数据传输用的指令。I/O 设备的地址也可位于存储空间内，这样可以利用整个指令系统和寻址方式的全部功能。对于高速的传输操作，8086 CPU 可以配置成最小模式，提供与传统的 DMA（Direct Memory Access，直接存储器存取）控制器兼容的信号线，还可配置成最大模式，与高性能通用 I/O 处理器 8089 配套使用。

1．输入/输出空间

8086 CPU 可访问的 I/O 空间所用的地址是 16 位的，这样有 64K 个 8 位端口或 32K 个 16 位端口。一个 8 位端口相当于一个存储器字节，分配 0000H～FFFFH 中的一个地址。任何两个相邻的 8 位端口可以组合成一个 16 位端口，并且类似存储器中的字。当程序访问偶端口的字时，需一次读（或写）操作；当访问奇端口的字时，则需两次读（或写）操作才能完成。

8086 CPU 提供专门的输入指令（IN）和输出指令（OUT），能够完成累加器（传输字节时是 AL，传输字时是 AX）和位于 I/O 空间的端口之间的数据传输。

8086 CPU 可以访问的 I/O 空间有 64 KB，可以满足绝大多数系统对 I/O 端口数目的要求，所以并不需要 I/O 空间像内存空间一样分段。8086 CPU 要访问某个端口，BIU 只需简单地把该端口的端口地址放到地址总线的低 16 位线上，就可以对该端口进行访问。

在 8086 CPU 指令系统中，有两种方式在 I/O 指令中指出端口地址：一种是把端口地址规定为在指令中的一个固定值，另一种是预先将端口地址送入 DX 寄存器中。

2．存储器编址的输入和输出

I/O 端口也可置于 8086 CPU 的存储空间内，只要这些设备能像存储器那样做出响应。存储器编址的输入和输出可以利用 8086 CPU 的指令系统和寻址方式的能力，进行输入和输出处理，且使程序设计更灵活。事实上，访问存储器的任何指令都能用来访问位于存储空间内的 I/O 端口，如传送指令 MOV 能在 8086 任意一个寄存器与一个端口之间传输数据，或者用“与”（AND）“或”（OR）等指令处理 I/O 设备寄存器中的各位。

8086 CPU 要为存储器编址的 I/O 付出代价：存储空间的一部分归 I/O 设备专用，减少了存储器的可用地址空间。实际上，在现代计算机系统中，对 I/O 端口的编址一般采用 CPU 设计时已给出的 I/O 访问方式，所以 8086 系统很少采用存储器编址的 I/O 方式对端口进行访问。

8086 CPU 为提供更高速的 I/O 数据传输功能，设计了满足 DMA 需要的引脚。当系统中有 DMA 控制器时，用于接管 CPU 对总线的控制权。在存储器与高速外设之间建立直接数据

块传输的高速通路，8086 CPU 配置成最小模式时，提供与 DMA 控制器兼容的 HOLD（保持）和 HLDA（保持响应）信号。DMA 控制器发出 HOLD 信号向 CPU 请求使用总线，如果 CPU 正在使用总线，将完成当前的总线周期（总线周期将在 2.3 节介绍），然后发出 HLDA 信号，将总线使用权交由 DMA 控制器管理。在 HOLD 信号变成无效前，CPU 不得使用总线。

2.2.3　8086 CPU 的最小模式和最大模式系统

以 8086 CPU 构成的微型计算机系统有最小模式和最大模式两种配置。最小模式是单机系统，系统中所需的控制信号全部由 8086 CPU 本身直接提供；最大模式可以构成多处理机系统，系统中所需的控制信号由总线控制器 8288 提供。CPU 工作模式的选择是由硬件决定的，当 CPU 的引脚 MN/$\overline{MX}$ 接高电平（+5 V）时，构成最小模式；当 MN/$\overline{MX}$ 接低电平（地）时，构成最大模式。两种不同模式下的主要区别是 8086 CPU 的第 24～31 号引脚具有不同的功能。

1．最小模式系统

8086 CPU 最小模式的基本配置如图 2-16 所示。其中，除了存储器、I/O 芯片和基本时钟发生器（8284A），还有用于地址的锁存器 8282（或 8283）、用于数据的缓冲器 8286（或 8287）。在 8086 系统中，地址线和数据线是复用的，所以地址锁存器是必要的。这些复用的引脚在某时刻只能体现地址线或数据线之一，所以在对存储器进行访问时，要先将地址输出。此时，复用的引脚是地址线，再利用地址锁存器保存这些地址。

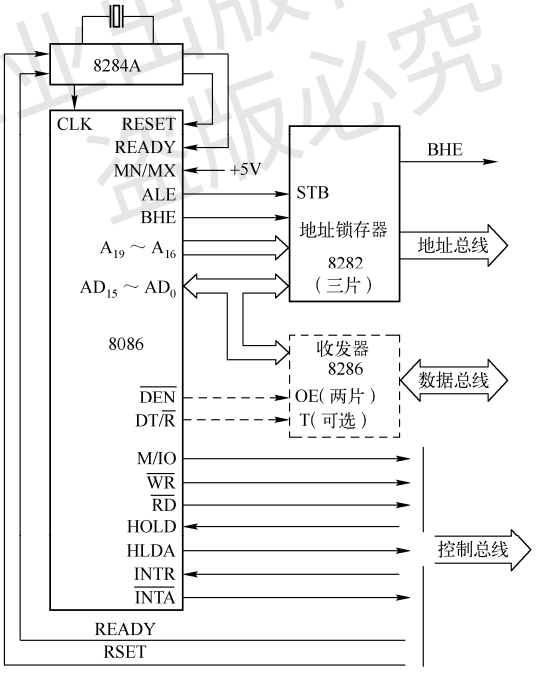

图 2-16　8086 CPU 最小模式

之后，这些引脚才是数据线，将数据读出或写到存储器。在某时刻，处理器把某个存储单元的地址发送到地址总线上，经锁存器将这些地址保存起来。只有这样，处理器才能把数据通过某些共享的引脚送到数据总线上，完成对存储器的读/写操作。从这个意义上讲，在 8086 最

小模式系统中，数据缓冲器就不是必要的。

8282（或 8283）是带三态缓冲器的通用 8 位数据锁存器。8282 的输入信号和输出信号是同相的，而 8283 的输入信号和输出信号是反相的。由于地址输出是单向的，因此选用单向的数据锁存器 8282 作为地址锁存。

8286（或 8287）是带三态输出的 8 位双向数据缓冲器，专用于需要双向传输的数据总线接口。8286 的输入或输出时信号是同相的，而 8287 的输入或输出时信号是反相的。在 8086 最小模式系统中，可以不用数据收发缓冲器 8286（或 8287），这时可将 CPU 的地址/数据总线直接与存储器或 I/O 端口的数据线相连。

最小模式系统还允许接入其他要求共享总线的设备。典型的例子是接入 DMA 控制器 8237 芯片，相关信号线是 HOLD 和 HLDA（2.2.2 节已介绍）。

2．最大模式系统

8086 CPU 最大模式的基本配置如图 2-17 所示，与最小模式系统相比，主要区别是最大模式系统中增加了总线控制器 8288 和总线仲裁器 8289。8086 CPU 输出的状态信号 S_2 ~ S_0 同时送给 8288 和 8289，由 8288 输出 8086 CPU 系统所需的控制信号，而 8289 总线仲裁器对系统中多个处理器提出共享总线资源的要求做出裁决。因此，8086 CPU 的最大模式由于 8288 和 8289 的存在，很容易构成一个多处理器系统。

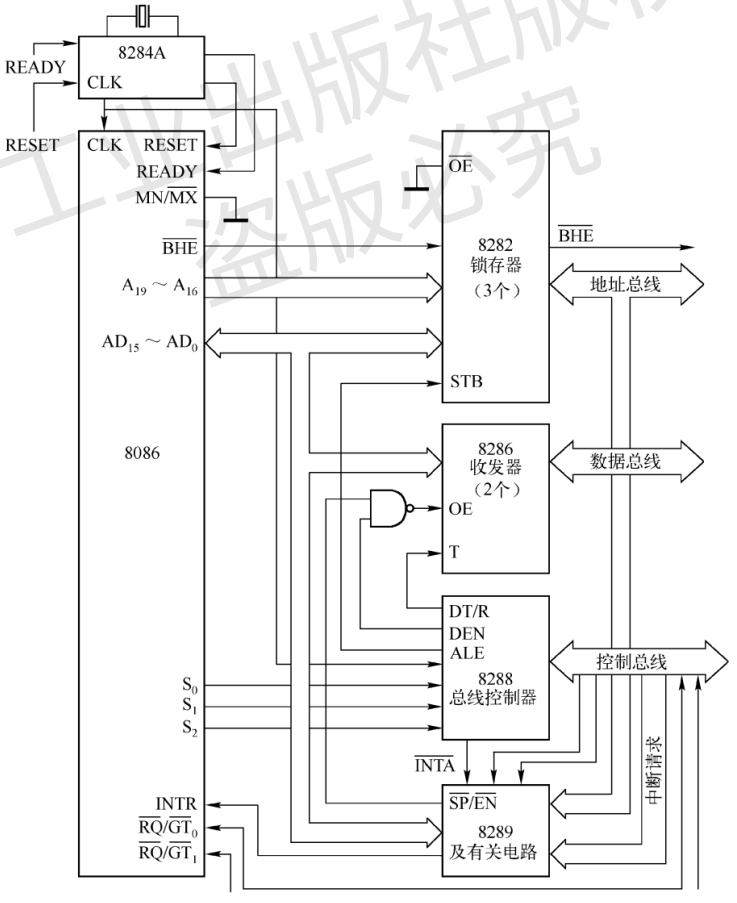

图 2-17　8086 CPU 最大模式

（1）总线控制器 8288

在最大模式中，8288 主要用来解释状态码 $\overline{S}_2$、$\overline{S}_1$、$\overline{S}_0$ 表示的总线状态，并产生 DT / $\overline{R}$ 、$\overline{DEN}$ 、MCE/$\overline{PDEN}$ 和 ALE 等一系列总线命令和控制信号。由 8288 产生的地址锁存允许信号 ALE、数据发送接收控制信号 DT / $\overline{R}$ 、数据传输允许信号 $\overline{DEN}$ 的功能与最小模式中直接由 8086 CPU 发出的控制信号相同，只是 $\overline{DEN}$ 信号的极性相反，因而系统中增设了一个非门。由 8086 CPU 输出的总线状态信号 $\overline{S}_2$、$\overline{S}_1$、$\overline{S}_0$ 与 8288 输出总线命令信号间的对应关系如表 2-9 所示。

表 2-9　8288 可提供的总线命令信号

总线状态信号			CPU 状态	8288 命令
$\overline{S}_2$	$\overline{S}_1$	$\overline{S}_0$		
0	0	0	中断响应	$\overline{INTA}$
0	0	1	读 I/O 端口	$\overline{IORC}$
0	1	0	写 I/O 端口	$\overline{IOWC}$ ， $\overline{AIOWC}$
0	1	1	暂停	无
1	0	0	取指令	$\overline{MRDC}$
1	0	1	读存储器	$\overline{MRDC}$
1	1	0	写存储器	$\overline{MWTC}$ ， $\overline{AMWC}$
1	1	1	无作用	无

（2）总线仲裁和总线仲裁器 8289

当 8086 构成最大模式系统时，HOLD 和 HLDA 引脚的信号演变成请求/同意信号 $\overline{RQ}$ / $\overline{GT}_0$ 和 $\overline{RQ}$ / $\overline{GT}_1$ 。这些双向信号传输引脚可以使 8086 和另外两个处理器共享局部总线。首先，要求使用总线的处理器通过 $\overline{RQ}$ / $\overline{GT}_0$ 或 $\overline{RQ}$ / $\overline{GT}_1$ 向 CPU 发出总线请求；当 CPU 接收这一请求后，即通过 $\overline{RQ}$ / $\overline{GT}_0$ 或 $\overline{RQ}$ / $\overline{GT}_1$ 回送总线同意信号。当占用总线的处理器用完总线后，它把一个脉冲发送给 8086，说明请求结束，CPU 这时可以收回对总线的控制权。共享总线的处理器数目一般是有限的，因为 8086 CPU 只提供两条请求/同意信号线，在硬件上保证 $\overline{RQ}$ / $\overline{GT}_0$ 比 $\overline{RQ}$ / $\overline{GT}_1$ 具有较高的优先级，因此当两个请求同时到达 CPU 时，8086 先应答通过 $\overline{RQ}$ / $\overline{GT}_0$ 引脚提出请求的处理器。当 CPU 正在处理 $\overline{RQ}$ / $\overline{GT}_1$ 上早先的请求时，来自 $\overline{RQ}$ / $\overline{GT}_0$ 的请求要等到 $\overline{RQ}$ / $\overline{GT}_1$ 的处理器释放对总线的控制权后，才能获得 CPU 的响应。

如果希望在一个系统中包含多个共享总线的处理器，即多处理器系统，就必须对总线进行仲裁，以保证优先级别高的处理器优先使用总线，这就需要一种协调各处理器使用共享总线的方法。总线仲裁器 8289 和总线控制器 8288 一起，为 8086 系统提供这种控制。

首先，在多处理器系统中，每个处理器（通常指主控者）必须配备一个 8288 总线控制器和一个 8289 总线仲裁器。总线上的每个处理器都分配不同的优先权。当总线请求同时到来时，8289 解决争用问题，经过仲裁把总线使用权转让给具有较高优先级的处理器。在解决总线争用的问题上，8289 采用 3 种优先权处理技术：并行优先权裁决方式、串行优先权裁决方式和循环优先权裁决方式（更详细的内容可查询 8289 总线仲裁器手册）。8289 对多处理器系统中每个处理器而言是透明的，每个处理器好像自己独占了总线一样。

2.3　8086 CPU 内部时序

各种微处理器的工作过程实际上就是执行指令的过程。它们所进行的操作是周期性的，即

取指令、执行指令、再取指令、再执行指令……由于传统的计算机对指令采用串行解释方式，因此总是一条指令执行完了再去取下一条指令，直到整个程序执行完毕。这种工作方式的优点是控制简单，指令之间不会产生任何关联，但速度慢，系统吞吐率比较低。

8086 CPU 因设置了可独立操作的指令执行部件（EU）和总线接口部件（BIU），且两者有明确的分工，所以系统的效率得到了提高。

为了更好地理解这一点，先来看 BIU 的工作情况。BIU 直接负责 CPU 与存储器和 I/O 端口交换数据，它的工作包括从存储器取出指令送入指令队列中，或者取出操作数去参加 EU 中的运算，或者将 EU 中的运算结果写入存储器中，而这些操作都要经过系统外部总线来完成。在 8086 CPU 中，把 BIU 完成一次访问存储器操作所需的时间称为一个总线周期。总线周期实际上就是一次访问存储器所需要的时间，即存储器的一个存取周期。在理想情况下，BIU 可处于连续工作状态，不断地访问存储器，或取指令，或读/写操作数。

再来看 EU 的工作情况。EU 负责执行指令，只需从指令队列中取得指令，并分析执行它。在指令执行过程中，可以根据需要随时要求 BIU 访问存储器，取操作数或写运算结果。关键的是，EU 的操作与 BIU 访问存储器的操作可以并行进行，因此在理想的情况下，EU 也可处于连续工作状态，不断地执行从指令队列中得到的指令。所谓“在理想情况下”，说明实际上 BIU 和 EU 不可能完全处于连续工作状态，因为 BIU 有可能由于 EU 执行某些复杂指令时内部操作时间很长，而不需要访问存储器，这时 BIU 处于空闲状态而不进入总线周期；另一方面，EU 有时需要等待 BIU 从存储器取出操作数后才能进行运算，尤其是遇到转移类指令时，就有可能使原来指令队列中已取出的指令全部作废，要等 BIU 重新从存储器中取出目标地址中的指令后，才能继续执行下条指令。EU 的内部操作过程可被 BIU 的总线周期覆盖，所以可以不考虑 EU 的内部操作时序。

在 8086 CPU 中，每个总线周期至少包含 4 个时钟周期（$T_1 \sim T_4$）。时钟周期是微处理器操作的最小单位，是由系统时钟的频率确定的。BIU 总是在 T_1 周期时将存储器的 20 位物理地址（或 16 位 I/O 端口地址）送上总线，在 $T_2 \sim T_4$ 周期期间，通过总线进行数据传输。

1．最小模式系统的读/写总线周期

（1）8086 CPU 读总线周期

最小模式下，8086 CPU 的读总线周期时序图如图 2-18 所示。

在 T_1 时钟周期，BIU 将被访问的存储单元的 20 位物理地址 $A_{19} \sim A_0$ 和总线高位有效信号 $\overline{BHE}$ 一起送到总线上。在地址锁存允许信号 ALE 的控制下，这些地址被锁存到 8282（或 8283）地址锁存器中，然后输出到地址总线上。以后由 $M/\overline{IO}$ 信号确定读存储器还是读 I/O 端口。

在 T_2 周期时，$AD_{15} \sim AD_0$ 成为高阻悬空状态，$A_{19}S_6 \sim A_{16}S_3$ 和 $\overline{BHE}/S_7$ 立即成为状态信息输出，与此同时，$\overline{RD}$ 信号有效，从而启动被选的存储器或 I/O 端口。

如果被选的存储器或 I/O 端口在 T_3 周期时来得及读出数据送到数据总线上，它们就将 READY 置为有效（高电平）。CPU 在 T_3 周期时钟脉冲的上升沿得知 READY 信号有效后，就在 T_3 周期结束时，在 $DT/\overline{R}=0$ 和 $\overline{DEN}=0$ 信号的控制下，将数据总线上的 16 位（或 8 位）有效数据经数据收发器 8286（或 8287）缓冲后向 CPU 输入，从而完成读存储器的任务。

如果配合工作的存储器或 I/O 端口由于本身速度或其他原因，来不及在 T_2 状态时读出所需信息，那么必须在 T_3 周期时钟脉冲的上升沿时使 READY 无效（低电平）。当 CPU 在此时得知 READY 信号无效时，就会在 T_3 周期之后插入一个等待周期 T_w，然后在 T_w 的时钟脉冲上升

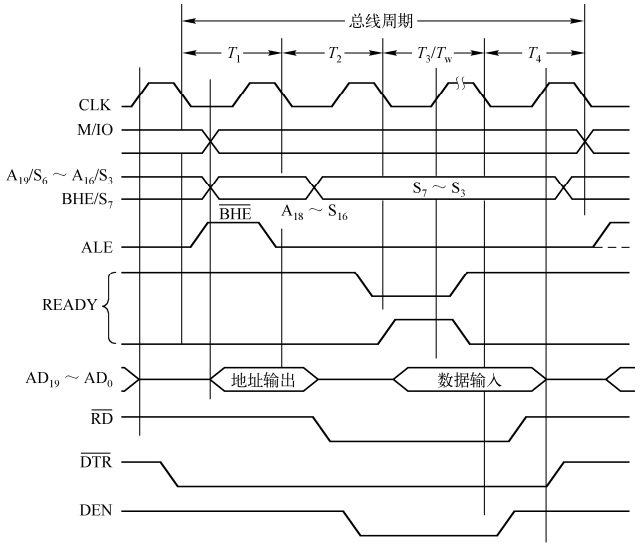

图 2-18　8086 CPU 的读总线周期时序图（最小模式）

沿再次测试 READY 信号，若还是无效则继续插入一个新的等待周期，直至 READY 有效为止。在插入 T_w 期间，其他控制信号保持同 T_3 状态时相同，在一个总线周期可插入若干个 T_w 周期，以协调高速 CPU 与低速存储器或 I/O 端口之间的数据传输。

（2）写总线周期

8086 CPU 在最小模式下的写总线周期时序图如图 2-19 所示，与读总线周期大同小异。下面主要说明它们的不同之处。

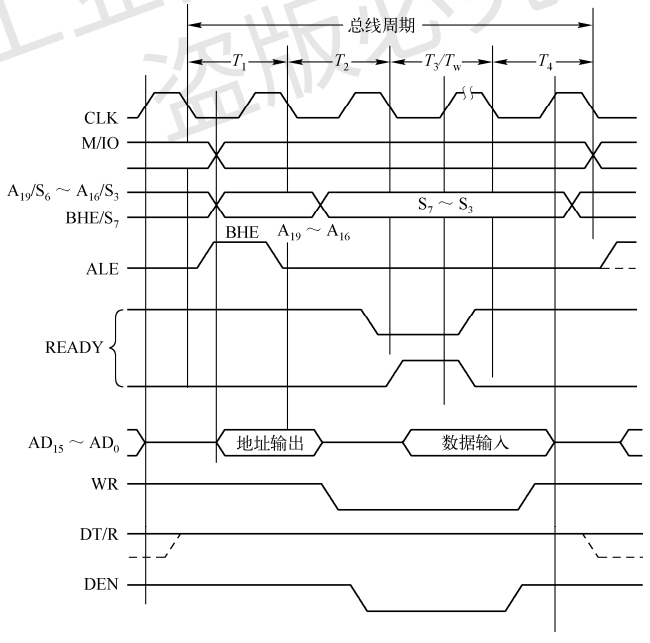

图 2-19　8086 CPU 的写总线周期时序图（最小模式）

在写总线周期，地址的传输过程与读总线周期完全相同，而当输出地址被锁存后，在地址数据复用总线 AD_{15} ~ AD_0 上会输出 16 位数据，同时当 T_w 有效时，向存储器或 I/O 端口发出写

命令，要求将数据总线上的数据写入指定的存储单元或 I/O 端口中去。在写总线周期中，DT / $\overline{R}$ 线应输出高电平，使数据缓冲器 8286（或 8287）呈输出状态。必要时，存储器或 I/O 端口可以通过 READY 信号要求 CPU 在 T_3 和 T_4 状态之间插入等待周期 T_w，以延长写入过程。

2．最大模式系统中 8086 CPU 的读/写总线周期

图 2-20 和图 2-21 分半为最大模式系统中 8086 CPU 的读和写总线周期时序图。

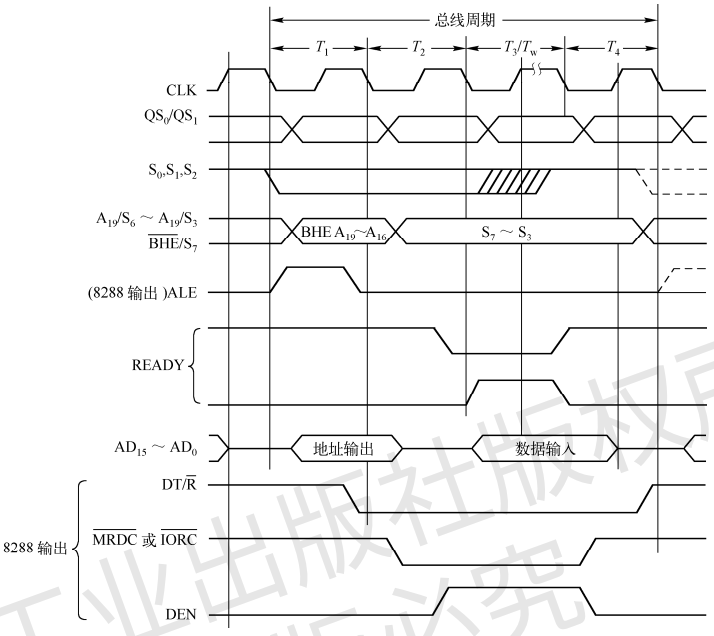

图 2-20　8086 CPU 的读总线周期时序图（最大模式）

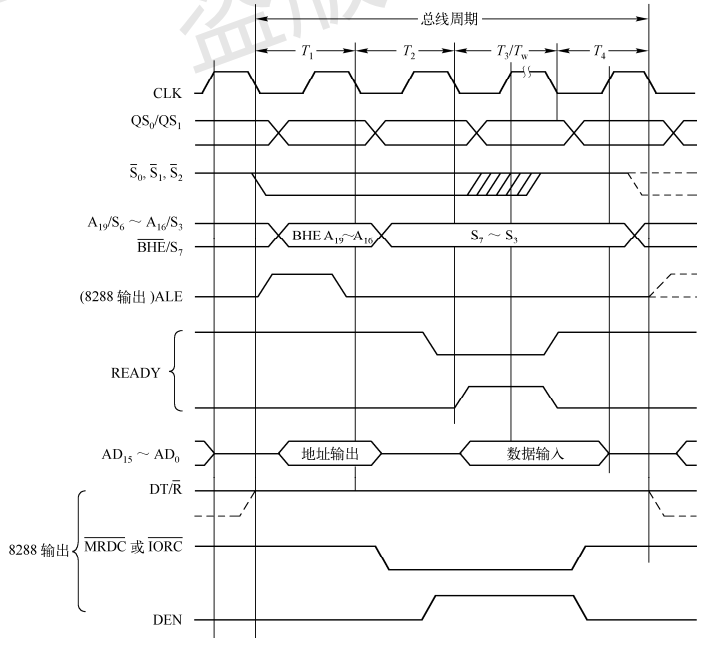

图 2-21　8086 CPU 的写总线周期时序图（最大模式）

同 8086 的最小模式相比，最大模式中增设了总线控制器 8288，因此有一些控制信号不再由 CPU 直接给出，而由 8288 依据 CPU 送来的三位状态信号 $\overline{S}_2$、$\overline{S}_1$ 和 $\overline{S}_0$ 译码后提供。

在图 2-20 和图 2-21 中，从总线周期的 T_1 时钟周期开始，CPU 就输出 $\overline{S}_2$、$\overline{S}_1$ 和 $\overline{S}_0$。8288 根据这些状态信息的不同编码分别输出对存储器或 I/O 端口的控制命令，完成读或写操作功能（详见表 2-9）。

习 题 2

1．8086 CPU 由哪两部分构成？它们的主要功能是什么？

2．8086 CPU 预取指令队列有什么好处？8086 CPU 内部的并行操作体现在哪里？

3．8086 CPU 中有哪些寄存器？各有什么用途？

4．下列情况下应判定哪个标志位并说明其状态。

（1）比较两个无符号数是否相等。

（2）两个无符号数相减后比较大小。

（3）两数运算后结果是正数还是负数。

（4）两数相加后是否产生溢出。

5．简述 8086 CPU 中物理地址的形成过程。8086 CPU 中的物理地址最多有多少个？逻辑地址呢？

6．8086 CPU 中的存储器为什么要采用分段结构？有什么好处？

7．在 8086 存储器中存放数据字时有“对准字”和“非对准字”之分，请说明它们的差别。

8．8086 CPU 工作在最小模式和最大模式系统中的主要区别是什么？各有什么主要特点？

9．在某系统中，已知当前(SS)=2360H，(SP)=0800H，请说明该堆栈段在存储器中的物理地址范围。若往堆栈中存放 20 字节数据，那么 SP 的内容为什么值?

10．已知当前数据段位于存储器的 B4000H～C3FFFH 范围内，则 DS 段寄存器的内容为多少？

11．8086 CPU 中为什么一定要有地址锁存器？需要锁存哪些信息？

12．8086 CPU 的读/写总线周期各包含多少个时钟周期？什么情况下需要插入等待周期 T_w？插入多少个 T_w 取决于什么因素？

13．若已知当前(DS)=7F06H，在偏移地址为 0075H 开始的存储器中连续存放 6 字节的数据，分别为 11H、22H、33H、44H、55H 和 66H。请指出这些数据在存储器中的物理地址。

14．某程序在当前数据段中存有两个数据字 0ABCDH 和 1234H，它们对应的物理地址分别为 3FF85H 和 40AFEH，若当前(DS)=3FB0H，请说明这两个数据字的偏移地址，并用图说明它们在存储器中的存放格式。