第3章 计算机的核心部件——微处理器

通过第 2 章的介绍可以看到，微处理器是整个计算机的重要组成部分。它是由一片或几片超大规模集成电路组成、具有运算器和控制器功能的中央处理部件，也称中央处理器(CPU)，有时也简称处理器(processor)。

随着大规模集成电路和超大规模集成电路工艺与技术的发展，微处理器的集成度越来越高，组成结构越来越复杂，功能也越来越强。图 3.1 给出了一个由 550 万晶体管构成的 Pentium pro(高能奔腾)微处理器的外观图示。

图 3.1　Pentium pro 微处理器的外观

本章首先介绍微处理器的工作模式、编程结构及寻址机制，然后介绍微处理器的内部组成、外部引脚信号及操作时序，以便对现代微处理器及整个计算机的结构和组成有进一步的了解和掌握。

3.1　微处理器的工作模式

为了既能发挥高性能 CPU 的处理能力，又可满足用户对应用软件兼容性的要求，自 80286 开始，出现了微处理器不同工作模式的概念。它较好地解决了 CPU 性能的提高与兼容性之间的矛盾。常见的微处理器工作模式有：实模式(real mode)、保护模式(protected mode)和虚拟 8086 模式(virtual 8086 mode)。

3.1.1　实模式

所谓实模式，简单地说就是 80286 以上的微处理器所采用的 8086 的工作模式。在实模式下，采用类似于 8086 的体系结构，其寻址机制、中断处理机制均和 8086 相同；物理地址的形成也同 8086 一样——将段寄存器的内容左移 4 位再与偏移地址相加(后面将详述)；寻址空间为 1MB，并采用分段方式，每段大小为 64KB；此外，在实模式下，存储器中保留两个专用区域，一个为初始化程序区：FFFF0H～FFFFFH，存放进入 ROM 引导程序的一条跳转指令；另一个为中断向量表区：00000H～003FFH，在这 1K 字节的存

储空间中可存放 256 个中断服务程序的入口地址，每个入口地址占 4 个字节，这与 8086 的情形相同。

实模式是 80x86 处理器在加电或复位后立即出现的工作方式，即使是想让系统运行在保护模式，系统初始化或引导程序也需要在实模式下运行，以便为保护模式所需要的数据结构做好各种配置和准备。也可以说，实模式是为建立保护式做准备的工作模式。

3.1.2 保护模式

保护模式是支持多任务的工作模式，它提供了一系列的保护机制，如任务地址空间的隔离、设置特权级、执行特权指令、进行访问权限的检查等。这些功能是实现 Windows 和 Linux 这样现代操作系统的基础。

80386 以上的微处理器在保护模式下可以访问 4G 字节的物理存储空间，段的长度在启动分页功能时是 4G 字节，不启动分页功能时是 1M 字节，分页功能是可选的。在这种方式下，可以引入虚拟存储器的概念，用以扩充编程者所使用的地址空间。

3.1.3 虚拟 8086 模式

虚拟 8086 模式又称“V86 模式”，是一种特殊的保护模式。它是既有保护功能又能执行 8086 代码的工作模式，是一种动态工作模式。在这种工作模式下，处理器能够迅速、反复进行 V86 模式和保护模式之间的切换，从保护模式进入 V86 模式执行 8086 程序，然后离开 V86 模式，进入保护模式继续执行原来的保护模式程序。

3.2 微处理器的编程结构

所谓微处理器的编程结构，即是在编程人员眼中看到的微处理器的软件结构模型。软件结构模型便于人们从软件的视角去了解计算机系统的操作和运行。从这一点上说，程序员可以不必知道微处理器内部极其复杂的电路结构、电气连接或开关特性，也不需要知道各个引脚上的信号功能和动作过程。对于编程人员来说，重要的是要了解微处理器所包含的各种寄存器的功能、操作和限制，以及在程序设计中如何使用它们。进一步，需要知道微处理器外部的存储器中数据是如何组织的，微处理器如何从存储器中取得指令和数据等。

3.2.1 程序可见寄存器

所谓“程序可见(program visible)寄存器”，是指在应用程序设计时可以直接访问的寄存器。相比之下，“程序不可见(program invisible)寄存器”是指在应用程序设计时不能直接访问，但在进行系统程序设计(如编写操作系统软件)时可以被间接引用或通过特

权指令才能访问的寄存器。在 80x86 微处理器系列中，通常在 80286 及其以上的微处理器中才包含程序不可见寄存器，主要用于保护模式下存储系统的管理和控制。

3.2.2 80x86/Pentium 处理器的寄存器模型

图 3.2 给出了 80x86/Pentium 微处理器的寄存器模型。它实际上是一个呈现在编程者面前的寄存器集合，所以也称微处理器的编程结构。早期的 8086/8088 及 80286 微处理器为 16 位结构，它们所包含的寄存器是图 3.2 所示寄存器集的一个子集；80386、80486 及 Pentium 系列微处理器为 32 位结构，它们包括了图 3.2 所示寄存器的全部。图中阴影区域的寄存器在 8086/8088 及 80286 微处理器中是没有的，它们是 80386、80486 及 Pentium 系列微处理器中新增加的。

由图 3.2 可见，该寄存器模型包括 8 位、16 位及 32 位寄存器组。8 位寄存器有 AH、AL、BH、BL、CH、CL、DH 和 DL，在指令中用双字母的寄存器名字来引用它们。例如“MOV AL,BL”指令，将 8 位寄存器 BL 的内容传送到 AL 中；16 位寄存器有 AX、BX、CX、DX、SP、BP、SI、DI、IP、FLAGS、CS、DS、ES、SS、FS 和 GS。这些寄存器也用双字母的名字来引用。例如，“MOV BX,CX”指令，将 16 位寄存器 CX 的内容传送到 BX 寄存器中；32 位寄存器为 EAX、EBX、ECX、EDX、ESP、EBP、EDI、ESI、EIP 和 EFLAGS。这些寄存器一般可用三字母的名字引用，例如“MOV EBX,ECX”指令，将 32 位寄存器 ECX 的内容传送到 EBX 中。

图 3.2 中的寄存器按功能的不同可分为通用寄存器、指令指针寄存器、标志寄存器和段寄存器 4 种类型。下面先对这些寄存器的基本功能予以概括说明，以便对它们有一个初步了解和认识。至于对这些寄存器的深入理解和正确使用，还需要一个过程，特别是通过后续章节中指令系统及汇编语言程序设计的学习过程。

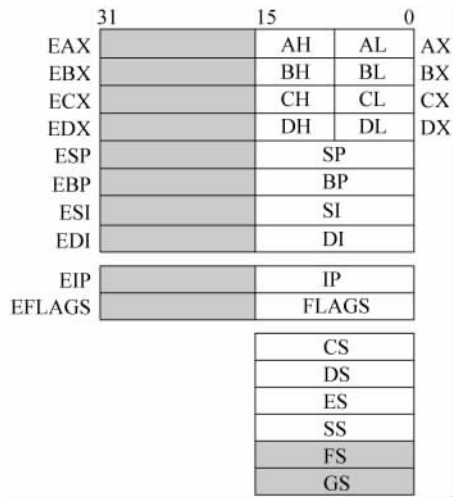


图 3.2 80x86/Pentium 处理器的寄存器模型

1. 通用寄存器

通用寄存器也称多功能寄存器，在图 3.2 所示的寄存器模型中，共有 8 个通用寄存器，即 EAX、EBX、ECX、EDX、ESP、EBP、ESI、EDI，按它们的功能差别，又可分为两组，即："通用数据寄存器"及"指针寄存器和变址寄存器"。

(1) 通用数据寄存器。通用数据寄存器用来存放 8 位、16 位或 32 位的操作数。大多数算术运算和逻辑运算指令都可以使用这些寄存器。共有 4 个通用数据寄存器，它们是 EAX、EBX、ECX 和 EDX。

EAX (Accumulator，累加器)：EAX 可以作为 32 位寄存器(EAX)、16 位寄存器(AX)或 8 位寄存器(AH 或 AL)引用。如果作为 8 位或 16 位寄存器引用，则只改变 32 位寄存器的一部分，其余部分不受影响。当累加器用于乘法、除法及一些调整指令时，它具有专门的用途，但通常仍称之为通用寄存器。在 80386 及更高型号的微处理器中，EAX 寄存器也可以用来存放访问存储单元的偏移地址。

EBX (Base，基址)：EBX 是个通用寄存器，它可以作为 32 位寄存器(EBX)、16 位寄存器(BX)或 8 位寄存器(BH 或 BL)引用。在 80x86 系列的各种型号微处理器中，均可以用 BX 存放访问存储单元的偏移地址。在 80386 及更高型号的微处理器中，EBX 也可以用于存放访问存储单元的偏移地址。

ECX (Count，计数)：ECX 是个通用寄存器，它可以作为 32 位寄存器(ECX)、16 位寄存器(CX)或 8 位寄存器(CH 或 CL)引用。ECX 可用来作为多种指令的计数值。用于计数的指令是重复的串操作指令、移位指令、循环移位指令和 LOOP/LOOPD 指令。移位和循环移位指令用 CL 计数，重复的串操作指令用 CX 计数，LOOP/LOOPD 指令用 CX 或 ECX 计数。在 80386 及更高型号的微处理器中，ECX 也可用来存放访问存储单元的偏移地址。

EDX (Data，数据)：EDX 是个通用寄存器，用于保存乘法运算产生的部分积，或除法运算之前的部分被除数。对于 80386 及更高型号的微处理器，这个寄存器也可用来寻址存储器数据。

(2) 指针寄存器和变址寄存器。这是另外 4 个通用寄存器，分别是：堆栈指针寄存器 ESP、基址指针寄存器 EBP、源变址寄存器 ESI 和目的变址寄存器 EDI。这 4 个寄存器均可作为 32 位寄存器引用 (ESP、EBP、ESI 和 EDI)，也可作为 16 位寄存器引用 (SP、BP、SI 和 DI)，主要用于堆栈操作和串操作中形成操作数的有效地址。其中，ESP、EBP (或 SP、BP)用于堆栈操作，ESI、EDI (或 SI、DI)用于串操作。另外，这 4 个寄存器也可作为数据寄存器使用。

ESP (Stack Pointer，堆栈指针)：ESP 寻址一个称为堆栈的存储区。通过这个指针存取堆栈存储器数据。具体操作将在本章后面介绍堆栈及其操作时再做说明。这个寄存器作为 16 位寄存器引用时，为 SP；作为 32 位寄存器引用时，则为 ESP。

EBP(Base Pointer，基址指针)：EBP 用来存放访问堆栈段的一个数据区的"基地址"。它作为 16 位寄存器引用时，为 BP；作为 32 位寄存器引用时，则为 EBP。

ESI(Source Index,源变址)：ESI 用于寻址串操作指令的源数据串。它的另一个功能是作为 32 位(ESI)或 16 位(SI)的数据寄存器使用。

EDI(Destination Index,目的变址)：EDI 用于寻址串操作指令的目的数据串。如同 ESI 一样,EDI 也可用为 32 位(EDI)或 16 位(DI)的数据寄存器使用。

2. 指令指针寄存器 EIP

EIP(Instruction Pointer)是一个专用寄存器,用于寻址当前需要取出的指令字节。当 CPU 从内存中取出一个指令字节后,EIP 就自动加 1,指向下一指令字节。当微处理器工作在实模式下时,这个寄存器为 IP(16 位)；当 80386 及更高型号的微处理器工作于保护模式下时,则为 EIP(32 位)。

程序员不能对 EIP/IP 进行存取操作。程序中的转移指令、返回指令以及中断处理能对 EIP/IP 进行操作。

3. 标志寄存器 EFLAGS

EFLAGS 用于指示微处理器的状态并控制它的操作。图 3.3 展示了 80x86/Pentium 系列所有型号微处理器的标志寄存器的情况。注意,从 8086/8088 到 Pentium Ⅱ 微处理器是向前兼容的。随着微处理器功能的增强及型号的更新,相应的标志寄存器的位数也不断扩充。早期的 8086/8088 微处理器的标志寄存器 FLAG 为 16 位,且只定义了其中的 9 位；80286 微处理器虽然仍为 16 位的标志寄存器,但定义的标志位已从原来的 9 位增加到 12 位(新增加了 3 个标志位)；80386 及更高型号的微处理器则采用 32 位的标志寄存器 EFLAGS,所定义的标志位也有相应的扩充。

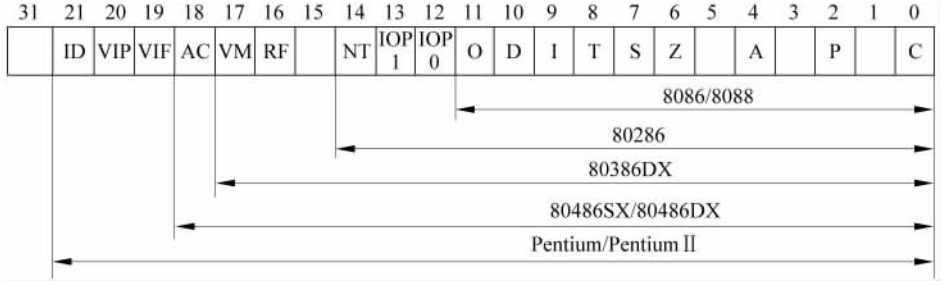


图 3.3　80x86/Pentium 系列微处理器的标志寄存器

下面着重介绍 8086/8088 系统中所定义的 9 个标志位——O、D、I、T、S、Z、A、P、C。在这 9 个标志位中,有 6 位(即 CF、PF、AF、ZF、SF 和 OF)为状态标志；其余 3 位(即 TF、IF 和 DF)为控制标志。状态标志与控制标志的作用有所不同。顾名思义,状态标志反映微处理器的工作状态,如执行加法运算时是否产生进位,执行减法运算时是否产生借位,运算结果是否为零等；控制标志对微处理器的运行起特定的控制作用,如以单步方式运行还是以连续方式运行,在程序执行过程中是否允许响应外部中断请求等。

6 个状态标志的功能简述如下：

(1) 进位标志 CF(Carry Flag)：运算过程中最高位有进位或借位时，CF 置 1；否则置 0。

(2) 奇偶标志 PF(Parity Flag)：该标志位反映运算结果低 8 位中 1 的个数情况，若为偶数个 1，则 PF 置 1；否则置 0。它是早期 Intel 微处理器在数据通信环境中校验数据的一种手段。今天，奇偶校验通常由数据存储和通信设备完成，而不是由微处理器完成。所以，这个标志位在现代程序设计中很少使用。

(3) 辅助进位标志 AF(Auxiliary carry Flag)：辅助进位标志也称“半进位”标志。若运算结果低 4 位中的最高位有进位或借位，则 AF 置 1；否则置 0。AF 一般用于 BCD 运算时是否进行十进制调整的依据。

(4) 零标志 ZF(Zero Flag)：反映运算结果是否为零。若结果为零，则 ZF 置 1；否则置 0。

(5) 符号标志 SF(Sign Flag)：记录运算结果的符号。若结果为负，则 SF 置 1；否则置 0。SF 的取值总是与运算结果的最高位相同。

(6) 溢出标志 OF(Overflow Flag)：反映有符号数运算结果是否发生溢出。若发生溢出，则 OF 置 1；否则置 0。所谓溢出，是指运算结果超出了计算装置所能表示的数值范围。例如，对于字节运算，数值表示范围为－128～＋127；对于字运算，数值表示范围为－32768～＋32767。若超过上述范围，则发生了溢出。溢出是一种差错，系统应做相应的处理。

在机器中，溢出标志的判断逻辑式为“OF＝最高位进位⊕次高位进位”。

注意：“溢出”与“进位”是两个不同的概念。某次运算结果有“溢出”，不一定有“进位”；反之，有“进位”，也不一定发生“溢出”。另外，“溢出”标志实际上是针对有符号数运算而言；对于无符号数运算，溢出标志 OF 是无定义的，无符号数运算的溢出状态可通过进位标志 CF 来反映。

下面，通过具体例子来进一步熟悉这 6 个状态标志的功能定义(为了便于表示，在例子中提前使用了第 4 章中介绍的 MOV 指令及 ADD 指令)。

【例 3.1】 指出执行如下指令后，标志寄存器中各状态标志值。

```
MOV  AX , 31C3H
ADD  AX , 5264H
```

上述两条指令执行后，在 CPU 中将完成如下二进制运算：

$$\begin{array}{rr} & 0011\ \ 0001\ \ 1100\ \ 0011 \\ + & 0101\ \ 0010\ \ 0110\ \ 0100 \\ \hline & 1000\ \ 0100\ \ 0010\ \ 0111 \end{array}$$

所以，根据前面给出的 6 个状态标志的功能定义，可得：

OF＝1 (最高位进位⊕次高位进位＝0⊕1＝1)；

SF＝1 (SF 与运算结果的最高位相同)；

ZF＝0 (运算结果不为零)；

AF＝0 (运算结果低 4 位中的最高位无进位)；

PF＝1 （运算结果低 8 位中 1 的个数为偶数）；

CF＝0 （运算结果最高位无进位）。

在本例中可以看到溢出标志 OF 和进位标志 CF 的情况。这里，OF＝1，说明有符号数运算时发生了溢出；但进位标志 CF＝0。

作为练习，请指出下述两条指令执行后的 6 个状态标志的情况：

```
MOV  AX , 0E125H
ADD  AX , 0C25DH
```

3 个控制标志的功能分述如下：

（1）方向标志 DF（Direction Flag）：用来控制串操作指令的执行。若 DF＝0，则串操作指令的地址自动增量修改，串数据的传送过程是从低地址到高地址的方向进行；若 DF＝1，则串操作指令的地址自动减量修改，串数据的传送过程是从高地址到低地址的方向进行。

（2）中断标志 IF（Interrupt Flag）：用来控制对外部可屏蔽中断请求的响应。若IF＝1，则 CPU 响应外部可屏蔽中断请求；若 IF＝0，则 CPU 不响应外部可屏蔽中断请求。

（3）陷阱标志 TF（Trap Flag）：陷阱标志也称单步标志。当 TF＝1 时，CPU 处于单步方式；当 TF＝0 时，则 CPU 处于连续方式。单步方式常用于程序的调试。

4. 段寄存器

由图 3.2 可以清楚地看到，微处理器寄存器集合中的另一组寄存器为 16 位的段寄存器，用于与微处理器中的其他寄存器联合生成存储器地址。80x86/Pentium 系列的微处理器中有 4 个或 6 个段寄存器。对于同一个微处理器而言，段寄存器的功能在实模式下和保护模式下是不相同的。本章主要针对实模式下段寄存器的基本功能进行介绍。下面概要列出每个段寄存器及其在系统中的功能：

（1）代码段寄存器（Code Segment，CS）：代码段是一个存储区域，用以保存微处理器使用的程序代码。代码段寄存器 CS 定义代码段的起始地址。

（2）数据段寄存器（Data Segment，DS）：数据段是包含程序所使用的大部分数据的存储区。与代码段寄存器 CS 类似，数据段寄存器 DS 用以定义数据段的起始地址。

（3）附加段寄存器（Extra Segment，ES）：附加段是为某些串操作指令存放目的操作数而附加的一个数据段。附加段寄存器 ES 用以定义附加段的起始地址。

（4）堆栈段寄存器（Stack Segment，SS）：堆栈是计算机存储器中的一个特殊存储区，用以暂时存放程序运行中的一些数据和地址信息。堆栈段寄存器 SS 定义堆栈段的首地址。通过堆栈段寄存器 SS 和堆栈指针寄存器 ESP/SP 可以访问堆栈栈顶的数据。另外，通过堆栈段寄存器 SS 和基址指针寄存器 EBP/BP 可以寻址堆栈栈顶下方的数据。具体的实现方法，将在后续章节介绍。

（5）段寄存器 FS 和 GS：这两个段寄存器仅对 80386 及更高型号的微处理器有效，以便程序访问相应的两个附加的存储器段。

3.3 微处理器的寻址机制

3.3.1 存储器分段技术

实模式下 80x86/Pentium 可直接寻址的地址空间为 $2^{20}=1$M 字节单元。这就是说，CPU 需输出 20 位地址信息才能实现对 1M 字节单元存储空间的寻址。但实模式下 CPU 中所使用的寄存器均是 16 位的，内部 ALU 也只能进行 16 位运算，其寻址范围局限在 $2^{16}=65536$(64K)字节单元。为了实现对 1M 字节单元的寻址，80x86 系统采用了存储器分段技术。具体做法是，将 1M 字节的存储空间分成许多逻辑段，每段最长 64K 字节单元，可以用 16 位地址码进行寻址。每个逻辑段在实际存储空间中的位置是可以浮动的，其起始地址可由段寄存器的内容来确定。实际上，段寄存器中存放的是段起始地址的高 16 位，称之为“段基值”(segment base value)。

80x86/Pentium 系列微处理器中设置了 4 个或 6 个 16 位的段寄存器。它们分别是代码段寄存器 CS、数据段寄存器 DS、附加段寄存器 ES、堆栈段寄存器 SS 以及在 80386 及更高型号微处理器中使用的段寄存器 FS 和 GS。

以 8086～80286 微处理器为例，由于设置有 4 个段寄存器，因此任何时候 CPU 可以定位当前可寻址的 4 个逻辑段，分别称为当前代码段、当前数据段、当前附加段和当前堆栈段。当前代码段的段基值(即段基地址的高 16 位)存放在 CS 寄存器中，该段存放程序的可执行指令；当前数据段的段基值存放在 DS 寄存器中，当前附加段的段基值存放在 ES 寄存器中，这两个段的存储空间存放程序中参加运算的操作数和运算结果；当前堆栈段的段基值存放在 SS 寄存器中，该段的存储空间用作程序执行时的存储器堆栈。

需要说明的是，各个逻辑段在实际的存储空间中可以完全分开，也可以部分重叠，甚至完全重叠。这种灵活的分段方式如图 3.4 所示。另外，段的起始地址的计算和分配通常是由系统完成的，并不需要普通用户参与。

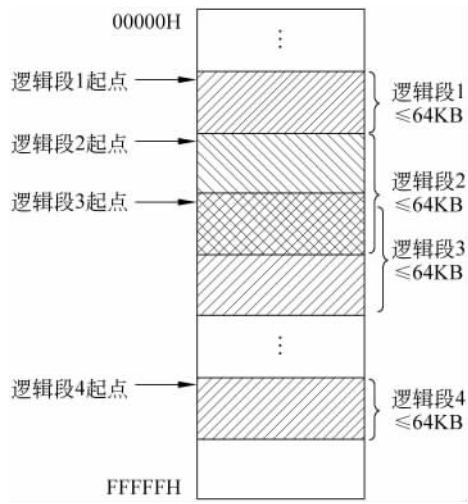


图 3.4 逻辑段在物理存储器中的位置

其实，还有其他方法也可以将 1M 字节单元的物理存储空间分成可用 16 位地址码寻址的逻辑段。例如将 20 位物理地址分成两部分：高 4 位为段号，可用机器内设置的 4 位长的“段号寄存器”来保存；低 16 位为段内地址，也称“偏移地址”，如图 3.5 所示。

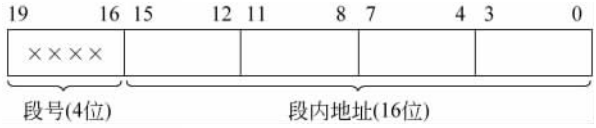


图 3.5　另一种分段方法

但是，这种分段方法有其不足之处。一是 4 位长的“段号寄存器”与其他寄存器不兼容，操作上会增添麻烦；二是这样划分的段每个逻辑段大小固定为 64K 字节单元，当程序中所需的存储空间不是 64K 字节单元的倍数时，就会浪费存储空间。反观前一种分段机制，则要灵活、方便得多，所以 80x86/Pentium 系统中采用了前一种分段机制。

3.3.2　实模式下的存储器寻址

1. 物理地址与逻辑地址

在有地址变换机构的计算机系统中，每个存储单元可以看成具有两种地址：物理地址和逻辑地址。物理地址是信息在存储器中实际存放的地址，它是 CPU 访问存储器时实际输出的地址。例如，实模式下的 80x86/Pentium 系统的物理地址是 20 位，存储空间为 $2^{20}=1$M 字节单元，地址范围从 00000H 到 FFFFFH。CPU 和存储器交换数据时所使用的就是这样的物理地址。

逻辑地址是编程时所使用的地址。或者说程序设计时所涉及的地址是逻辑地址而不是物理地址。编程时不需要知道产生的代码或数据在存储器中的具体物理位置。这样可以简化存储资源的动态管理。在实模式下的软件结构中，逻辑地址由“段基值”和“偏移量”两部分构成。前面已提及，“段基值”是段的起始地址的高 16 位。“偏移量”(offset)也称偏移地址，它是所访问的存储单元距段的起始地址之间的字节距离。给定段基值和偏移量，就可以在存储器中寻址所访问的存储单元。

在实模式下，“段基值”和“偏移量”均是 16 位的。“段基值”由段寄存器 CS、DS、SS、ES、FS 和 GS 提供；“偏移量”由 BX、BP、SP、SI、DI、IP 或以这些寄存器的组合形式来提供。

2. 实模式下物理地址的产生

实模式下 CPU 访问存储器时的 20 位物理地址可由逻辑地址转换而来。具体方法是，将段寄存器中的 16 位“段基值”左移 4 位(低位补 0)，再与 16 位的“偏移量”相加，即可得到所访问存储单元的物理地址，如图 3.6 所示。

上述由逻辑地址转换为物理地址的过程也可以表示成如下计算公式：

物理地址＝段基值×16＋偏移量

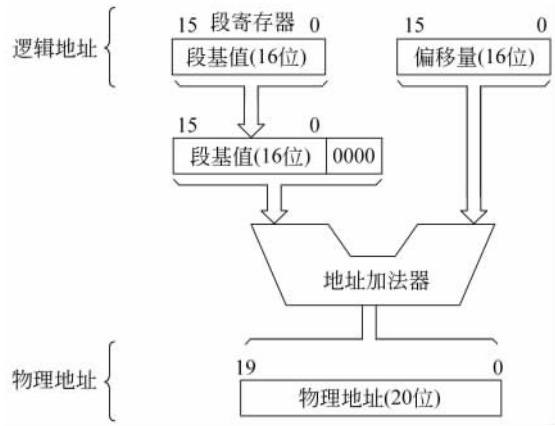
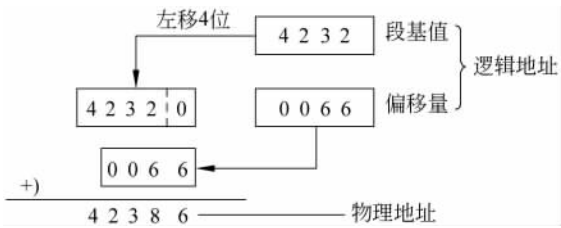


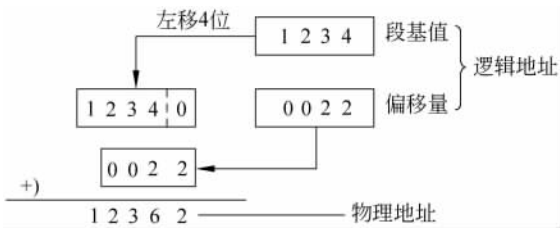
图 3.6　实模式下物理地址的产生

上式中的“段基值×16”在微处理器中是通过将段寄存器的内容左移 4 位(低位补 0)来实现的，与偏移量相加的操作则由地址加法器来完成。

【例 3.2】　设代码段寄存器 CS 的内容为 4232H，指令指针寄存器 IP 的内容为 0066H，即 CS＝4232H，IP＝0066H，则访问代码段存储单元的物理地址计算如下：



【例 3.3】　设数据段寄存器 DS 的内容为 1234H，基址寄存器 BX 的内容为 0022H，即 DS＝1234H，BX＝0022H，则访问数据段存储单元的物理地址计算如下：



【例 3.4】　若段寄存器内容是 002AH，产生的物理地址是 002C3H，则偏移量是多少？

解　将段寄存器内容左移 4 位，低位补 0 得：002A0H。

从物理地址中减去上列值得偏移量为：002C3H－002A0H＝0023H。

需注意的是，每个存储单元有唯一的物理地址，但它可以由不同的“段基值”和“偏移量”转换而来，这只要把段基值和偏移量改变为相应的值即可。也就是说，同一个物理地址与多个逻辑地址相对应。例如，段基值为 0020H，偏移量为 0013H，构成的物理地址为 00213H；然而，若段基值改变为 0021H，配以新的偏移量 0003H，其物理地址仍然是 00213H，如图 3.7 所示。

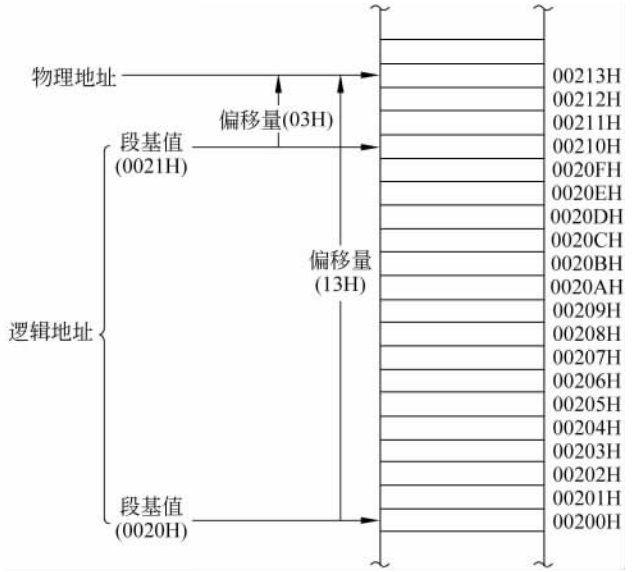


图 3.7　一个物理地址对应多个逻辑地址

3. “段加偏移”寻址

上述由段基值(段寄存器的内容)和偏移量相结合的存储器寻址机制也称为“段加偏移”寻址机制，所访问的存储单元的地址常被表示成“段基值：偏移量”的形式。例如，若段基值为 2000H，偏移量为 3000H，则所访问的存储单元的地址为 2000H ：3000H。

图 3.8 进一步说明了这种“段加偏移”的寻址机制如何选择所访问的存储单元的情形。这里段寄存器的内容为 1000H，偏移地址为 2000H。图中显示了一个 64KB 长的存储器段，该段起始于 10000H，结束于 1FFFFH。图中也表示了如何通过段基值(段寄存器的内容)和偏移量找到存储器中被选单元的情形。偏移量也称偏移地址，如图中所示，它是自段的起始位置到所选存储单元之间的字节距离。

图 3.8 中段的起始地址 10000H 是由段寄存器内容 1000H 左移 4 位低位补 0(或在 1000H 后边添加 0H)而得到的。段的结束地址 1FFFFH 是由段起始地址 10000H 与段长度 FFFFH(64K)相加的结果。

还需指出的是，在这种“段加偏移”的寻址机制中，由于是将段寄存器的内容左移 4 位(相当于乘以十进制数 16) 来作为段的起始地址的，所以实模下各个逻辑段只能起始

于存储器中 16 字节整数倍的边界。这样可以简化实模式下 CPU 生成物理地址的操作。通常称这 16 字节的小存储区域为“分段”或“节”(paragraph)。

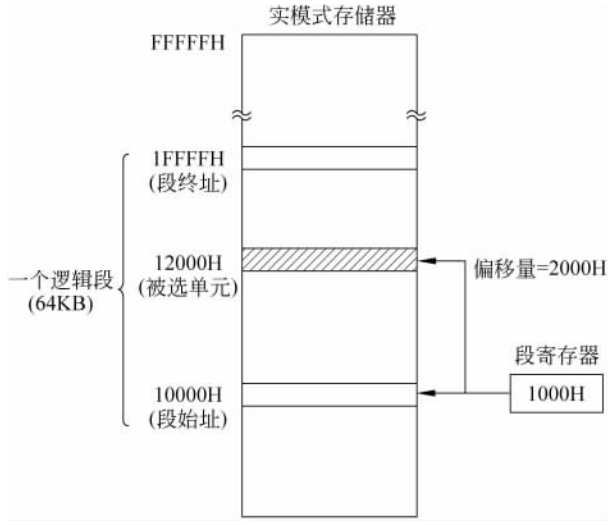


图 3.8　实模式下存储器寻址机制——“段加偏移”

4. 默认的段和偏移寄存器

在“段加偏移”的寻址机制中,微处理器有一套用于定义各种寻址方式中段寄存器和偏移地址寄存器的组合规则。例如,代码段寄存器总是和指令指针寄存器组合用于寻址程序的一条指令。这种组合是 CS: IP 还是 CS: EIP 取决于微处理器的操作模式。代码段寄存器定义代码段的起点,指令指针寄存器指示代码段内指令的位置。这种组合(CS: IP 或 CS: EIP)定位微处理器执行的下一条指令。例如,若 CS=2000H,IP=3000H,则微处理器从存储器的 2000H:3000H 单元,即 23000H 单元取下一条指令。

8086～80286 微处理器各种默认的 16 位“段加偏移”寻址组合方法如表 3-1 所示。表 3-2 表示 80386 及更高型号的微处理器使用 32 位“段加偏移”寻址组合的默认情况。80386 及更高型号的微处理器的“段加偏移”寻址组合比 8086～80286 微处理器的选择范围更大。

表 3-1　默认的 16 位“段加偏移”寻址组合

寄存器	偏移地址寄存器	主要用途
CS	IP	指令地址
SS	SP 或 BP	堆栈地址
DS	BX、DI、SI、8 位或 16 位数	数据地址
ES	操作指令的 DI	串操作目的地址

表 3-2　默认的 32 位“段加偏移”寻址组合

段寄存器	偏移地址寄存器	主 要 用 途
CS	EIP	指令地址
SS	ESP 或 EBP	堆栈地址
DS	EAX、EBX、ECX、EDX、EDI、ESI、8 位(16 位或 32 位)数	数据地址
ES	串操作指令的 EDI	串操作目的地址
FS	无默认	一般地址
GS	无默认	一般地址

5.“段加偏移”寻址机制允许程序重定位

“段加偏移”寻址机制给系统带来的一个突出优点就是允许程序或数据在存储器中重定位。重定位是程序或数据的一种重要特性,它是指一个完整的程序或数据块可以在有效的存储空间中任意地浮动并重新定位到一个新的地址区域。

这是由于在现代计算机的寻址机制中引入了分段的概念之后,用于存放段地址的段寄存器的内容可以由程序重新设置,所以在偏移地址不变的情况下,就可以将整个程序或数据块移动到存储器任何新的可寻址区域去。例如,一条指令位于距段首(段起始地址)6 个字节的位置,它的偏移地址是 6。当整个程序移到新的区域时,这个偏移地址 6 仍然指向距新的段首 6 个字节的位置,只是段寄存器的内容必须重新设置为程序所在的新存储区的地址。如果没有这种重定位特性,一个程序在移动之前必须大范围地重写或改写,这要花费大量时间,且容易出现差错。

“段加偏移”寻址机制所带来的这种可重定位特性,使编写与具体位置无关的程序(动态浮动码)成为可能。

3.3.3　堆栈

堆栈是存储器中的一个特定的存储区,它的一端(栈底)是固定的,另一端(栈顶)是浮动的,信息的存入和取出都只能在浮动的一端进行,并且遵循后进先出(Last-In First-Out)的原则。堆栈主要用来暂时保存程序运行时的一些地址或数据信息。例如,当 CPU 执行调用(Call)指令时,用堆栈保存程序的返回地址(亦称断点地址);在中断响应及中断处理时,通过堆栈“保存现场”和“恢复现场”;有时也利用堆栈为子程序传递参数。

堆栈是在存储器中实现的,并由堆栈段寄存器 SS 和堆栈指针寄存器 SP 来定位。SS 寄存器中存放的是堆栈段的段基值,它确定了堆栈段的起始位置。SP 寄存器中存放的是堆栈操作单元的偏移量,SP 总是指向栈顶。图 3.9 给出了堆栈的基本结构及操作示意图。值得注意的是,这种结构的堆栈是所谓“向下生长的”,即栈底在堆栈的高地址端,当堆栈为空时 SP 就指向栈底。因此,堆栈段的段基址(由 SS 寄存器确定)并不是栈底。

实模式下的堆栈为 16 位宽(字宽),堆栈操作指令(PUSH 指令或 POP 指令)对堆栈的操作总是以字为单位进行。即要压栈(执行 PUSH 指令)时,先将 SP 的值减 2,然后将

16 位的信息压入新的栈顶；要弹栈(执行 POP 指令)时，先从当前栈顶取出 16 位的信息，然后将 SP 的值加 2。可概括为："压栈时，先修改栈指针后压入"，"弹栈时，先弹出后修改栈指针"。

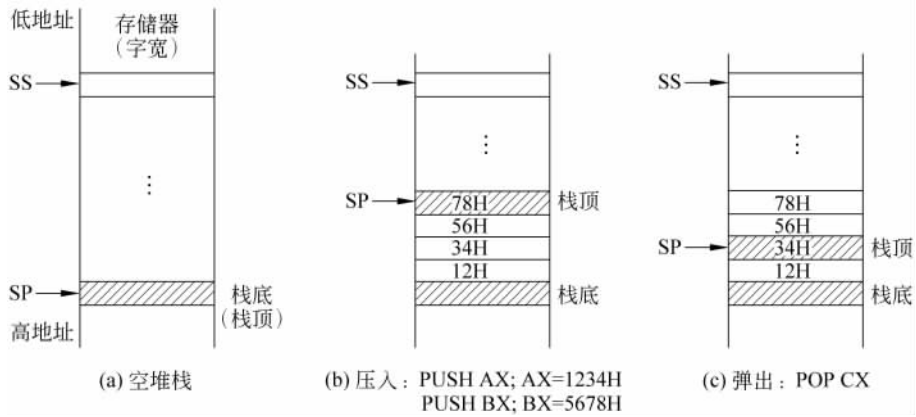


图 3.9　堆栈的结构与操作

由图 3.9 可以看出，堆栈的操作既不对堆栈中的项进行移动，也不清除它们。压栈时在新栈顶写入信息，弹栈时则只是简单地改变 SP 的值指向新的栈顶。

3.4　微处理器的内部组成结构及相关技术

为了说明现代微处理器的内部组成结构，我们给出一个经适当简化的 Pentium 处理器的内部结构框图(如图 3.10 所示)，并以此为例对现代微处理器的主要组成部件及其实现技术做概要说明。

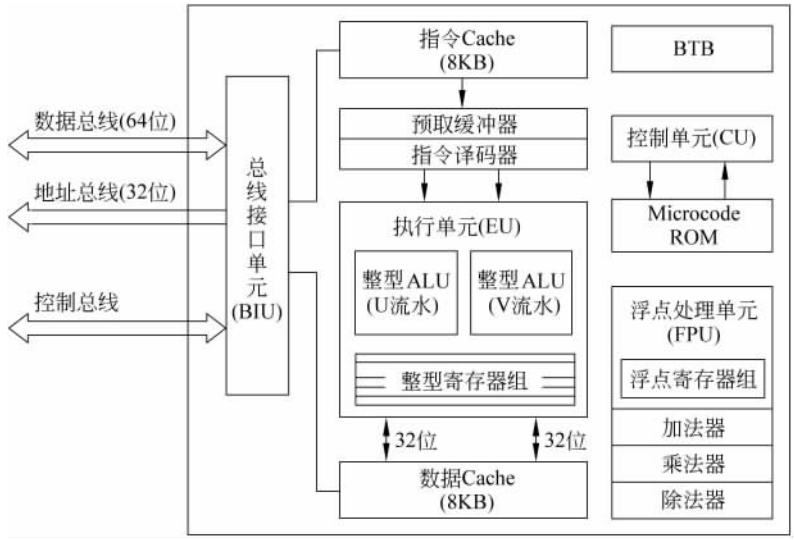


图 3.10　Pentium 处理器的内部结构框图

3.4.1 总线接口单元 BIU

总线接口单元(Bus Interface Unit,BIU)是微处理器与微机中其他部件(如存储器、I/O 接口等)进行连接与通信的物理界面。通过这个界面,实现微处理器与其他部件之间的数据信息、地址信息以及控制命令信号的传送。由图 3.10 可见,Pentium 处理器的外部数据总线宽度为 64 位,它与存储器之间的数据传输率可达 528MB/s。但需要说明的是,由于 Pentium 处理器内部的算术逻辑单元 ALU 和寄存器的宽度仍是 32 位的,所以它仍属于 32 位微处理器。

从图 3.10 还可以看到,Pentium 处理器的地址总线位数为 32 位,即它的直接寻址物理地址空间为 $2^{32}=4$GB。另外,BIU 还有地址总线驱动、数据总线驱动、总线周期控制及总线仲裁等多项功能。

3.4.2 指令 Cache 与数据 Cache

Cache(高速缓存)技术是现代微处理器及微型计算机设计中普遍采用的一项重要技术,它可以使 CPU 在较低速的存储器件条件下获得较高速的存储器访问,并提高系统的性能价格比。在 Pentium 之前的 80386 设计中,曾在处理器外部设置一个容量较小但速度较快的“片外 Cache”;而在 80486 中,则是在处理器内部设置了一个 8KB 的“片内 Cache”,统一作为指令和数据共用的高速缓存。

Pentium 处理器中的 Cache 设计与 80386 和 80486 有很大的不同,它采用哈佛结构,即把 Cache 分为“指令 Cache”和“数据 Cache”分别设置,从而避免仅仅设置统一 Cache 时发生存储器访问冲突的现象。Pentium 包括两个 8KB 的 Cache,一个为 8KB 的数据 Cache,一个为 8KB 的指令 Cache。指令 Cache 只存储指令,而数据 Cache 只存储指令所需的数据。

在只有统一的高速缓存的微处理器(如 80486)中,一个数据密集的程序很快就会占满高速缓存,几乎没有空间用于指令缓存,这就降低了微处理器的执行速度。而在 Pentium 中就不会发生这种情况,因为它有单独的指令 Cache。如图 3.10 所示,经过 BIU,指令被保存在 8KB 的“指令 Cache”中,而指令所需要的数据则保存在 8KB 的“数据 Cache”中。这两个 Cache 可以并行工作,并被称为“一级 Cache”或“片内 Cache”,以区别于设置在微处理器外部的“二级 Cache”或“片外 Cache”。

为了进一步提高计算机的性能,目前在高性能微处理器片内也采用 Cache 分级结构,具有一级 Cache、二级 Cache,有些微处理器(如安腾系列微处理器)片内还有三级 Cache 或四级 Cache。

3.4.3 超标量流水线结构

“超标量流水线”结构是 Pentium 处理器设计技术的核心。为了说明其特点,我们先

简要说明微处理器中“流水线”方式的概念，然后简要介绍“超标量”及“超级流水线”的技术特点。

流水线(pipeline)方式是把一个重复的过程分解为若干子过程，每个子过程可以与其他子过程并行进行的工作方式。由于这种工作方式与工厂中生产流水线十分相似，因此称为流水线技术。采用流水线技术设计的微处理器，把每条指令分为若干个顺序的操作(如取指、译码、执行等)，每个操作分别由不同的处理部件(如取指部件、译码部件、执行部件等)来完成。这样构成的微处理器，可以同时处理多条指令。而对于每个处理部件来说，每条指令的同类操作(如取指令)就像流水一样连续被加工处理。这种指令重叠、处理部件连续工作的计算机(或处理器)，称为流水线计算机(或处理器)。

采用流水线技术，可加快计算机执行程序的速度并提高处理部件的使用效率。图 3.11 表示了把指令划分为 5 个操作步骤并由处理器中 5 个处理部件分别处理时流水线的工作情形。

图 3.11　5 级流水的工作情形

如图 3.11 所示，流水线中的各个处理部件可并行工作，从而可使整个程序的执行时间缩短。容易看到，在图中所示的 7 个时间单位内，已全部执行完 3 条指令。如果以完全串行的方式执行，则 3 条指令需 3×5＝15 个时间单位才能完成。显然，采用流水线方式可以显著提高计算机的处理速度。

Pentium 处理器的流水线由分别称为“U 流水”和“V 流水”的两条指令流水线构成(双流水线结构)，其中每条流水线都拥有自己的地址生成逻辑、ALU 及数据 Cache 接口。因此，Pentium 处理器可以在一个时钟周期内同时发送两条指令进入流水线。比相同频率的单条流水线结构(如 80486)性能提高了一倍。通常称这种具有两条或两条以上能够并行工作的流水线结构为超标量(superscalar)结构。

与图 3.11 所示的情形相同，Pentium 的每一条流水线也是分为 5 个阶段(5 级流水)，即“指令预取”、“指令译码”、“地址生成”、“指令执行”和“回写”。当一条指令完成预取步骤时，流水线就可以开始对另一条指令的操作和处理。这就是说，Pentium 处理器实现的是两条流水线的并行操作，而每条流水线由 5 个流水级构成。

另外，还可以将流水线的若干流水级进一步细分为更多的阶段(流水小级)，并通过一定的流水线调度和控制，使每个细分后的“流水小级”可以与其他指令的不同的“流水小级”并行执行，从而进一步提高微处理器的性能。这被称为“超级流水线”技术(superpipelining)。

“超级流水线”与上面介绍的“超标量”结构有所不同，超标量结构是通过重复设置多个“取指”部件，设置多个“译码”、“地址生成”、“执行”和“写结果”部件，并让这些功能部件同时工作来加快程序的执行，实际上是以增加硬件资源为代价来换取处理器性能的；而超级流水线处理器则不同，它只需增加少量硬件，是通过各部分硬件的充分重叠工作来提高处理器性能的。从流水线的时空角度上看，超标量处理器主要采用的是空间并行性，而超级流水线处理器主要采用的是时间并行性。

从超大规模集成电路(VLSI)的实现工艺来看，超标量处理器能够更好地适应 VLSI 工艺的要求。通常，超标量处理器要使用更多的晶体管，而超流水线处理器则需要更快的晶体管及更精确的电路设计。

为了进一步提高处理器执行指令的并行度，可以把超标量技术与超流水线技术结合在一起，这就是“超标量超流水线”处理器。例如，Intel 的 P6 结构(Pentium Ⅱ/Ⅲ处理器)就是采用这种技术的更高性能微处理器，其超标度为 3(即有 3 条流水线并行操作)，流水线的级数为 12 级。

3.4.4 动态转移预测及转移目标缓冲器 BTB

正是由于计算机指令中具有能够改变程序流向的指令，才使得程序结构灵活多样，程序功能丰富多彩。这类指令一般包括跳转(JMP)指令、调用(CALL)指令和返回(RET)指令等，统称为转移(branch)指令。转移指令又可分为“无条件转移指令”及“条件转移指令”两大类。无条件转移指令执行时一定会发生转移，而条件转移指令执行时是否发生转移则取决于指令所要求的条件当时是否满足。例如，80x86 系统中的条件转移指令“JC START”，执行时若进位标志 CF 为 1，则使程序转移到“转移目标地址”START 处；否则，将顺序执行紧接着这条指令之后的下一条指令。

然而，转移指令也给处理器的流水线操作带来麻烦。因为在处理器预取指令时还未对指令进行译码，即它还不知道哪条指令是转移指令，所以只能按程序的静态顺序进行。也就是说，即使是遇到一条转移指令，也无法到“转移目标地址”处去预取指令装入指令队列，而只能顺序地装入紧接着转移指令之后的若干条指令。而当指令被执行并确实发生转移时，指令预取缓冲器中预先装入的指令就没用了。此时必须将指令缓冲器中原来预取的指令废除(也称“排空”流水线)，并从转移目标地址开始处重新取指令装入流水线。这样就极大地影响了流水线的处理速度和性能。

已有多项技术用于减小转移指令对流水线性能的影响，如基于编译软件的“延迟转移”(delayed branching) 技术和基于硬件的“转移预测”(branch prediction)技术。转移预测又有“静态转移预测”及“动态转移预测”之分。静态转移预测只依据转移指令的类型来预测转移是否发生。例如，对某一类条件转移指令总是预测为转移发生，对另一类总是预测转移不发生。显然，静态转移预测的正确率不会很高，只能作为其他转移处理

技术的辅助手段。动态转移预测法(dynamic branch prediction)是依据一条转移指令过去的行为来预测该指令的将来行为。即处理器要有一个“不断学习”的过程。由于程序结构中有众多重复或循环执行的机会,所以在预测算法选得较好的情况下,动态转移预测会达到较高的正确率,故被现代微处理器所普遍采用。下面,仍以 Pentium 为例来简要说明这种动态转移预测法的基本工作原理。

从图 3.12 可以看到,Pentium 提供了一个称为“转移目标缓冲器”BTB(Branch Target Buffer)的小 Cache 来动态预测程序的转移操作。在程序执行时,若某条指令导致转移,便记忆下这条转移指令的地址及转移目标地址(放入 BTB 内部的“登记项”中),并用这些信息来预测这条指令再次发生转移时的路径,预先从这里记录的“转移目标地址”处预取指令,以保证流水线的指令预取不会空置。其基本工作机制如图 3.12 所示。

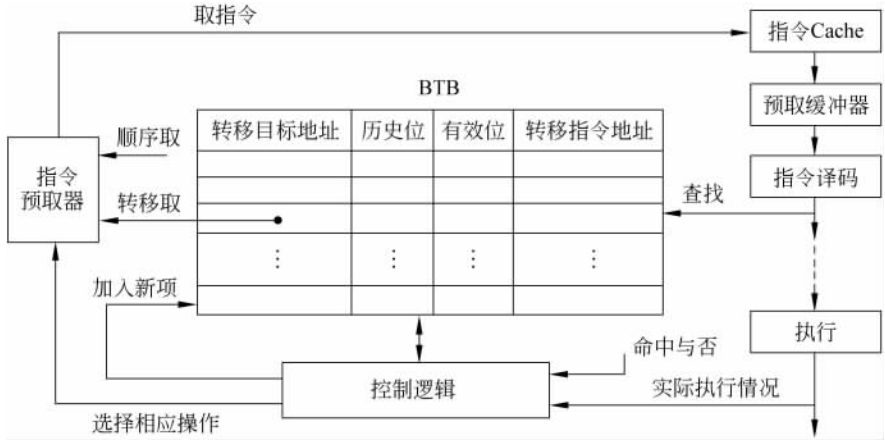


图 3.12 Pentium BTB 的工作机制[7]

“指令译码”阶段检查从预取缓冲器中取出的指令是否为转移指令,若是转移指令,则将此指令的地址送往 BTB 进行查找。若 BTB 命中(即在 BTB 中存在相应的登记项),则根据该项的“历史位”状态预测此指令在执行阶段是否发生转移。若预测为发生转移,则将该项中登记的“转移目标地址”提交给指令预取器,并指挥指令预取器从“转移目标地址”处提取指令装入预取缓冲器,即进行图 3.12 中所示的“转移取”;若预测为不发生转移,则从该转移指令的下一条指令开始提取指令,即进行所谓“顺序取”。若 BTB 未命中(即在 BTB 中不存在相应的登记项),则说明 BTB 中没有该指令的历史记录,此时固定预测为不发生转移,即固定进行“顺序取”。至于该指令在执行阶段实际发生转移时的处理情况,将在下面介绍 Pentium“执行单元”的功能时再作具体说明。

BTB 登记项中的“历史位”用以登记相应转移指令先前的执行行为,并用于预测此指令执行时是否发生转移。在执行阶段要根据实际是否发生转移,来修改命中项的历史位;或对于 BTB 未命中的转移指令而在执行阶段发生转移的情况,在 BTB 中建立新项

(加入新项)并设定历史位为 11。图 3.13 给出了 BTB 历史位的意义及状态转换情况。

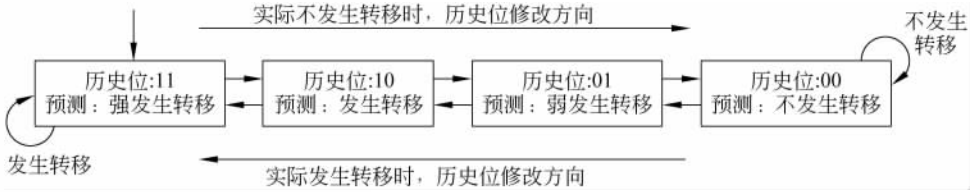


图 3.13　Pentium BTB 历史位的意义及状态转换[7]

由图 3.13 可以看出，Pentium 对历史位意义的设定更倾向于预测转移发生。历史位 11 常称为“强发生”(strongly taken)状态，10 称为“发生”(taken)状态，01 称为“弱发生”(weakly taken)状态，这 3 种历史位都预测转移发生。

后来的 Pentium 系列处理器使用更多的历史位，以更精细的转移预测算法来降低预测失误率。例如 Pentium 4 使用 4 位历史位，能在转移预测时考虑到更长的历史状况，能够显著地降低预测失误率。

另外，容易想到，对于循环程序而言，在首次进入循环和退出循环时，都将出现转移预测错误的情况。即首次进入循环时，预测不发生转移，而实际发生转移；退出循环时，预测发生转移，而实际不发生转移。这两种情况下均需要重新计算转移地址，并造成流水线的停顿和等待。但若循环 10 次，2 次预测错误而 8 次正确；循环 100 次，2 次预测错误而 98 次正确。因此，循环次数越多，BTB 的效益越明显。

3.4.5　指令预取器和预取缓冲器

指令预取器总是按给定的指令地址，从指令 Cache 中顺序地取出指令放入预取缓冲器中，直到在指令译码阶段遇到一条转移指令并预测它在指令执行阶段将发生转移时为止。此时，如图 3.12 所示，由 BTB 提供预测转移发生时的目标地址，并按此地址开始再次顺序地取指令，直到又遇到一条转移指令并预测转移发生时为止。指令预取器就是以这种折线式顺序由指令 Cache 取出指令装入预取缓冲器的。

3.4.6　指令译码器

指令译码器的基本功能是将预先取来的指令进行译码，以确定该指令的操作。

Pentium 处理器中，指令译码器的工作过程可分为两个阶段，在第一个阶段，对指令的操作码进行译码，并检查是否为转移指令。若是转移指令，则将此指令的地址送往 BTB。再进一步检查 BTB 中该指令的历史记录，并决定是否实施相应的转移预测操作；在第二个阶段，指令译码器需生成存储器操作数的地址。在保护方式下，还需按保护

模式的规定检查是否有违约地址，若有，则产生“异常”(exception)，并进行相应的处理。

3.4.7 执行单元 EU

指令的执行以两个 ALU 为中心，完成 U、V 流水线中两条指令的算术及逻辑运算。执行单元的主要功能如下：

① 按地址生成阶段(即指令译码的第二阶段)提供的存储器操作数地址，首先在 1 级数据 Cache 中获取操作数，若 1 级数据 Cache“未命中”(操作数未在 Cache 中)，则在 2 级 Cache(片外 Cache)或主存中查找。总之，在指令执行阶段的前半部，指令所需的存储器操作数、寄存器操作数要全部就绪，接着在指令执行阶段的后半部完成指令所要求的算术及逻辑操作。

② 确认在指令译码阶段对转移指令的转移预测是否与实际情况相符，即确认预测是否正确。若预测正确，则除了适当修改 BTB 中的“历史位”外，其他什么事情也不发生；若预测错误，则除了修改“历史位”外，还要清除该指令之后已在 U、V 流水线中的全部指令(“排空”流水线)，并指挥“指令预取器”重新取指令装入流水线。

另外，对于前面提到的在查找 BTB 时“未命中”从而固定预测为不发生转移的情况，若在执行阶段此指令确实没有发生转移，则其他什么事情也不发生，以后再遇到此转移令时仍作为一个“新面孔”的转移指令按前述办法来对待；如果在执行阶段此指令实际发生转移的话，则按“预测错误”处理，此时除了“排空”流水线外，还需将“转移目标地址”提交给 BTB，连同在指令译码阶段提交的“转移指令地址”，在 BTB 中建立一个新项，并设定“历史位”为“强发生”状态(11)。

3.4.8 浮点处理单元 FPU

顾名思义，浮点处理单元(Floating Point Unit，FPU)专门用来处理浮点数或进行浮点运算，因此也称浮点运算器。在 8086、80286 及 80386 年代，曾设置单独的 FPU 芯片(8087、80287 和 80387)，并称为算术协处理器(Mathematical Coprocessor)，简称协处理器。那时的主板上配有专门的协处理器插座。自从 80486 DX 开始，则将 FPU 移至微处理器内部，成为微处理器芯片的一个重要组成部分(如图 3.10 所示)。

Pentium 处理器的 FPU 性能已做了很大改进。FPU 内有 8 个 80 位的浮点寄存器 FR0～FR7，内部数据总线宽度为 80 位，并有分立的浮点加法器、浮点乘法器和浮点除法器，可同时进行 3 种不同的运算。

FPU 的浮点指令流水线也是双流水线结构。每条流水线分为 8 个流水级：预取指令、指令译码、地址生成、取操作数、执行 1、执行 2、写回结果和错误报告。

3.4.9 控制单元 CU

控制单元(Control Unit,CU)的基本功能是控制整个微处理器按照一定的时序过程一步一步地完成指令的操作。Pentium 的大多数简单指令都是以“硬布线”方式来实现的,如 2.1.2 节所述,采用这种方式,指令通过“指令译码器”译码后结合特定的时序条件即可产生相应的微操作控制信号,从而控制指令的执行,它可以获得较快的指令执行速度;而对于那些复杂指令的执行则是以“微程序”方式实现的。按照微程序实现方式,是将指令执行时所需要的微操作控制信号变成相应的一组微指令并预先存放在一个只读存储器中,当指令执行时,按安排好的顺序从只读存储器中一条一条读出这些微指令,从而产生相应的微操作控制信号去控制指令的执行。

“微程序”方式与“硬连线”方式是 CPU 控制指令执行的两种不同的实现方式。它们各有不同的特点。一般来说,“微程序”方式较方便灵活,但指令执行速度较慢,在传统的微处理器设计如 CISC(Complex Instruction Set Computer)结构中常被采用;“硬连线”方式灵活性较差,但它的突出优点是指令执行速度很快,常用于 RISC (Reduced Instruction Set Computer)结构的机器中。也可以说,RISC 结构中一般不使用“微程序”技术。

另外,控制单元还负责流水线的时序控制,以及处理与“异常”和“中断”有关的操作和控制。

3.5 微处理器的外部功能特性

为了更好地理解和使用现代微处理器,还应对其外部引脚信号及其操作特性有必要的了解。在本节,将以 32 位微处理器 80386 DX 为例,详细介绍微处理器外部引脚的基本功能特性及其操作时序。

3.5.1 微处理器的外部引脚信号

1. 80386 DX 的外部引脚信号概况

80386 DX 微处理器共 132 个外部引脚,用来实现与存储器、I/O 接口或其他外部电路进行连接和通信。整个芯片采用引脚栅格阵列(Pin Grid Array,PGA)封装,引脚分布如图 3.14 所示。

按功能的不同,可将这 132 个引脚信号分成 4 组:存储器/IO 接口、中断接口、DMA 接口和协处理器接口。图 3.15 给出了 80386 DX 引脚信号分组情况。

表 3-3 列出了各个引脚信号的名称、功能、传送方向以及每个信号的有效电平。例

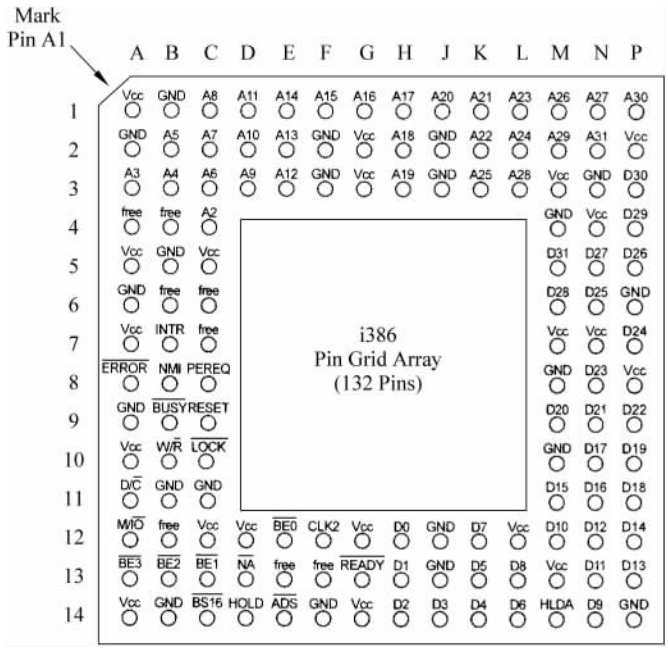


图 3.14　80386 DX 引脚分布

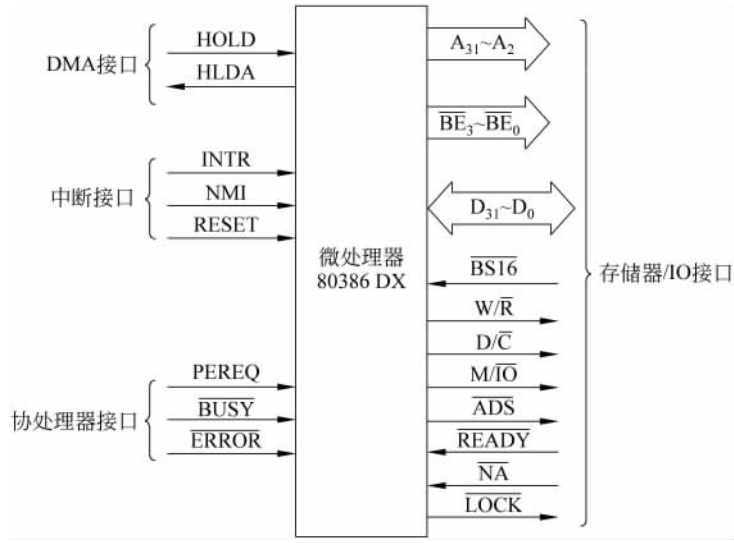


图 3.15　80386 DX 引脚信号分组

如,“存储器/IO 接口”中的 M/$\overline{IO}$信号,其功能是“存储器/IO 指示”,用以告诉外部电路当前微处理器是在访问存储器还是 I/O 接口；该信号的传送方向是输出,即它是由微处理器产生的输出信号；它的有效电平为 1/0,其含义为,在这个信号线上的逻辑 1 电平表明 CPU 当前是在访问存储器,而逻辑 0 电平表明是在访问 I/O 接口。又如,“中断接口”

中的 INTR 信号，是可屏蔽中断请求输入信号，其有效电平是逻辑 1。外部设备利用这个信号通知微处理器，它们需要得到服务。

表 3-3　80386 DX 外部引脚信号列表

名　　称	功　　能	方　　向	有 效 电 平
CLK2	系统时钟	输入	—
$A_{31}\sim A_2$	地址总线	输出	—
$\overline{BE_3}\sim\overline{BE_0}$	字节允许	输出	0
$D_{31}\sim D_0$	数据总线	输入/输出	—
$\overline{BS16}$	16 位总线宽	输入	0
$W/\overline{R}$	写/读指示	输出	1/0
$D/\overline{C}$	数据/控制指示	输出	1/0
$M/\overline{IO}$	存储器/IO 指示	输出	1/0
$\overline{ADS}$	地址状态	输出	0
$\overline{READY}$	就绪	输入	0
$\overline{NA}$	下一地址请求	输入	0
$\overline{LOCK}$	总线封锁	输出	0
INTR	中断请求	输入	1
NMI	非屏蔽中断请求	输入	1
RESET	系统复位	输入	1
HOLD	总线保持请求	输入	1
HLDA	总线保持响应	输出	1
PEREQ	协处理器请求	输入	1
$\overline{BUSY}$	协处理器忙	输入	0
$\overline{ERROR}$	协处理器错	输入	0

2. 存储器/IO 接口信号

微处理器的“存储器/IO 接口”信号通常又包括地址总线、数据总线及其他有关控制信号。下面分别予以说明。

1）地址和数据总线信号

地址总线和数据总线形成了 CPU 与存储器和 I/O 子系统间进行通信的基本通路。在早期的 Intel 微处理器（如 8085、8086/8088）中，曾普遍采用地址总线和数据总线复用技术，即将部分（或全部）地址总线与数据总线共用微处理器的一部分引脚，目的是为了减少微处理器的引脚数量，但由此也会带来控制逻辑及操作时序上的复杂性。自 80286 及更高型号的微处理器开始，则采用分开的地址和数据总线。如图 3.15 所示，80386 DX 的地址总线信号 $A_{31}\sim A_2$ 和数据总线信号 $D_{31}\sim D_0$ 被分别设定在不同的引脚上。

从硬件的观点来看，80386 DX 的实模式与保护模式之间仅有一点不同，即地址总线的规模。在实模式下，只输出低 18 位地址信号 $A_{19}\sim A_2$；而在保护模式下，则输出 30 位地址信号 $A_{31}\sim A_2$。其实，实模式的地址长度为 20 位，保护模式的地址长度是 32 位。其

余的两位地址码 A_1 和 A_0 被 80386 DX 内部译码，产生字节允许信号$\overline{BE_3}$、$\overline{BE_2}$、$\overline{BE_1}$ 和$\overline{BE_0}$，以控制在总线上进行字节、字或双字数据传送。

由图 3.15 及表 3-3 可以看到，地址总线是输出信号线。它们用于传送从 CPU 到存储器或 I/O 接口的地址信息。在实模式下，20 位地址给出了 80386 DX 寻址 1M(2^{20})字节物理地址空间的能力；而在保护模式下，32 位地址可以寻址 4G(2^{32})字节的物理地址空间。

无论是在实模式下还是保护模式下，80386 DX 微型计算机均具有独立的 I/O 地址空间。该 I/O 地址空间的大小为 64K 字节单元。所以，在寻址 I/O 设备时，仅需使用地址线 A_{15}～A_2 及相应的字节允许信号$\overline{BE_3}$、$\overline{BE_2}$、$\overline{BE_1}$ 和$\overline{BE_0}$。

数据总线由 32 条数据线 D_{31}～D_0 构成。由图 3.15 及表 3-3 可以看到，数据总线是双向的，即数据既可以由存储器或 I/O 接口输入给 CPU，也可以由 CPU 输出给存储器或 I/O 接口。在数据总线上传送数据的类型是对存储器读/写的数据或指令代码、对外部设备输入/输出的数据以及来自中断控制器的中断类型码等。

如前所述，在一个总线周期内，80386 DX 在数据总线上可以传送字节、字或双字。所以，它必须通知外部电路发生何种形式的数据传送以及数据将通过数据总线的哪一部分进行传送。80386 DX 是通过激活相应的字节允许信号($\overline{BE_3}$～$\overline{BE_0}$)来做到这一点的。表 3-4 列出了每个字节允许信号及对应被允许的数据总线部分。

表 3-4　字节允许及数据总线信号

字 节 允 许	数据总线信号
$\overline{BE_0}$	D_7～D_0
$\overline{BE_1}$	D_{15}～D_8
$\overline{BE_2}$	D_{23}～D_{16}
$\overline{BE_3}$	D_{31}～D_{24}

【例 3.5】 当字节允许信号$\overline{BE_3}$ $\overline{BE_2}$ $\overline{BE_1}$ $\overline{BE_0}$＝1100 时，将产生哪种类型的数据传送(字节、字、双字)？数据传送经过哪些数据线？

解　由表 3-4 容易发现，此时将在数据线 D_{15}～D_0 上进行一个数据字的传送。

2) 控制信号

微处理器的控制信号用来支持和控制在地址和数据总线上进行的信息传输。通过这些控制信号表明，何时有效地址出现在地址总线上，数据以什么样的方向在数据总线上传送，写入到存储器或 I/O 接口的数据何时在数据总线上有效，以及从存储器或 I/O 接口读出的数据何时能够在数据总线上放好，等等。

80386 DX 并不直接产生上述功能的控制信号，而是在每个总线周期的开始时刻输出总线周期定义的指示信号。这些总线周期指示信号需在外部电路中进行译码，从而产生对存储器和 I/O 接口的控制信号。

3 个信号用来标识 80386 DX 的总线周期类型，即在图 3.15 及表 3-3 中所列出的“写/读指示”(W/$\overline{R}$)、“数据/控制指示”(D/$\overline{C}$)及“存储器/IO 指示”(M/$\overline{IO}$)信号。表 3-5

列出了这些总线周期指示信号的全部状态组合及对应的总线周期类型。

表 3-5　总线周期指示信号及总线周期类型

$M/\overline{IO}$	$D/\overline{C}$	$W/\overline{R}$	总线周期类型
0	0	0	中断响应
0	0	1	空闲
0	1	0	读 I/O 数据
0	1	1	写 I/O 数据
1	0	0	读存储器代码
1	0	1	暂停/关机
1	1	0	读存储器数据
1	1	1	写存储器数据

由表 3-5 可见，$M/\overline{IO}$的逻辑电平标识是产生存储器还是 I/O 总线周期，逻辑 1 表明是存储器操作，而逻辑 0 则是 I/O 操作；$D/\overline{C}$ 标识当前的总线周期是数据还是控制总线周期。从表中可见，该信号的逻辑 0 电平表明是中断响应、读存储器代码以及暂停/关机操作的控制总线周期，而逻辑 1 电平表明是对存储器及 I/O 端口进行读/写操作的数据总线周期。仔细观察表 3-5 可以发现，若 $M/\overline{IO}$和 $D/\overline{C}$ 的编码是 00，则一个中断请求被响应；如果是 01，则进行 I/O 操作；如果是 10，则读出指令代码；如果是 11，则读/写存储器数据。

表 3-5 中的 $W/\overline{R}$ 信号用来标识总线周期的操作类型。若在一个总线周期中 $W/\overline{R}$ 为逻辑 0，则数据从存储器或 I/O 接口读出；相反，若 $W/\overline{R}$ 为逻辑 1，则数据被写入存储器或 I/O 接口。

在表 3-5 中，总线周期指示码 $M/\overline{IO}$、$D/\overline{C}$、$W/\overline{R}$＝001 的总线周期类型为空闲(idle)，这是一种不形成任何总线操作的总线周期，也称空闲周期。

【例 3.6】 若总线周期指示码 $M/\overline{IO}$、$D/\overline{C}$、$W/\overline{R}$＝010，则将产生什么类型的总线周期？

解　从表 3-5 不难发现，总线周期指示码 010 标识着一个“读 I/O 数据”的总线周期。

在图 3.15 的“存储器/IO 接口”中，还可以看到另外 3 个控制信号，即地址状态($\overline{ADS}$)、就绪($\overline{READY}$)及下一地址($\overline{NA}$)信号。$\overline{ADS}$为逻辑 0 表示总线周期指示码($M/\overline{IO}$、$D/\overline{C}$、$W/\overline{R}$)、字节允许信号($\overline{BE_3}\sim\overline{BE_0}$)及地址信号($A_{31}\sim A_2$)全为有效状态。

$\overline{READY}$信号用于插入等待状态(T_w)到当前总线周期中，以便通过增加时钟周期数使总线周期得到扩展。在图 3.15 中可以看到，这个信号是输入给 80386 DX 的。通常它是由存储器或 I/O 子系统产生并经外部总线控制逻辑电路提供给 80386 DX。通过将$\overline{READY}$信号变为逻辑 0，存储器或 I/O 接口可以告诉 80386 DX 它们已经准备好，处理器可以完成数据传送操作。关于这方面的操作特性，在下面介绍微处理器的操作时序时还会具体讨论。

还需指出，80386 DX 支持在其总线接口上的地址流水线方式。所谓地址流水线，是指对下一个总线周期的地址、总线周期指示码及有关的控制信号可以在本总线周期结束

之前发出，从而使对下一个总线周期的寻址与本总线周期的数据传送相重叠。采用这种方式，可以用较低速的存储器电路获得与较高速存储器相同的性能。外部总线控制逻辑电路是通过将$\overline{NA}$输入信号有效（变为逻辑 0）来激活这种流水线方式的。

由 80386 DX 输出的另一个控制信号是总线封锁（$\overline{LOCK}$）信号。这个信号用以支持多处理器结构。在使用共享资源（如全局存储器）的多处理器系统中，该信号能够用来确保系统总线和共享资源的占用不被间断。当微处理器执行带有$\overline{LOCK}$前缀的指令时，则$\overline{LOCK}$输出引脚变为逻辑 0，从而可以封锁共享资源以独占使用。

最后一个控制信号是“16 位总线宽”（$\overline{BS16}$）输入信号。该信号用来选择 32 位（$\overline{BS16}$＝1）或 16 位（$\overline{BS16}$＝0）数据总线。在实际应用中，如果 80386 DX 大多数情况下工作在 16 位数据总线方式，则可索性选用微处理器 80386 SX，它的数据总线宽度为 16 位。

3. 中断接口信号

由图 3.15 可见，80386 DX 的中断接口信号有“中断请求”（INTR）、“非屏蔽中断请求”（NMI）及“系统复位”（RESET）。INTR 是一个对 80386 DX 的输入信号，用来表明外部设备需要得到服务。80386 DX 在每条指令的开始时刻采样这个输入信号。INTR 引脚上的逻辑 1 电平表示出现了中断请求。

当 80386 DX 检测到有效的中断请求信号后，它便把这一事实通知给外部电路并启动一个中断响应总线周期时序。在表 3-5 中可以看到，中断响应总线周期的出现是通过总线周期指示码 M/$\overline{IO}$、D/$\overline{C}$、W/$\overline{R}$ 等于 000 来通知外部电路的。这个总线周期指示码将被外部总线控制逻辑电路译码从而产生一个中断响应信号。通过这个中断响应信号，80386 DX 告诉发出中断请求的外部设备它的服务请求已得到同意。这样就完成了中断请求和中断响应的握手过程。从此时开始，程序控制转移到了中断服务程序。

INTR 输入是可屏蔽的，即它的操作可以通过微处理器内部的标志寄存器中的“中断标志位”（IF）予以允许或禁止。而非屏蔽中断 NMI 输入，顾名思义，它是不可屏蔽的中断输入。只要在 NMI 引脚上出现 0 到 1 的跳变，不管中断标志 IF 的状态如何，一个中断服务请求总会被微处理器所接受。在执行完当前指令后，程序一定会转移到非屏蔽中断服务程序的入口处。

最后，RESET 输入用来对 80386 DX 进行硬件复位。例如，利用这个输入可以使微型计算机在加电时被复位。RESET 信号跳变到逻辑 1，将初始化微处理器的内部寄存器。当它返回到逻辑 0 时，程序控制被转移到系统复位服务程序的入口处。该服务程序用来初始化其余的系统资源，如 I/O 端口、中断标志及数据存储器等。执行 80386 DX 的诊断程序也是复位过程的一部分。它可以确保微型计算机系统的有序启动。

4. DMA 接口信号

由图 3.15 可见，80386 DX 的直接存储器访问（Direct Memory Access，DMA）接口只通过两个信号实现：总线保持请求（HOLD）和总线保持响应（HLDA）。

当一个外部电路（如 DMA 控制器）希望掌握总线控制权时，它就通过将 HOLD 输入

信号变为逻辑 1 来通知当前的总线主 80386 DX。80386 DX 如果同意放弃总线控制权(未在执行带 LOCK 前缀的指令),就在执行完当前总线周期后,使相关的总线输出信号全部变为高阻态(第三态),并通过将 HLDA 输出信号变到逻辑 1 电平来通知外部电路它已交出了总线控制权。这样就完成了"总线保持请求"和"总线保持响应"的握手过程。80386 DX 维持这种状态直至"总线保持请求"信号撤销(变为逻辑 0),随之 80386 DX 将"总线保持响应"信号也变为逻辑 0,并重新收回总线控制权。

5. 协处理器接口信号

在图 3.15 中可以看到,在 80386 DX 微处理器上提供了协处理器接口信号,以实现与数值协处理器 80387 DX 的接口。80387 DX 不能独立地形成经数据总线的数据传送。每当 80387 DX 需要从存储器读或写操作数时,它必须通知 80386 DX 来启动这个数据传送过程。这是通过将 80386 DX 的"协处理器请求"(PEREQ)输入信号变为逻辑 1 来实现的。

另外两个协处理器接口信号是$\overline{\text{BUSY}}$和$\overline{\text{ERROR}}$。"协处理器忙"($\overline{\text{BUSY}}$)是 80386 DX 的一个输入信号。每当协处理器 80387 DX 正在执行一条数值运算指令时,它就通过将$\overline{\text{BUSY}}$输入信号变为逻辑 0 来通知 80386 DX。另外,如果在协处理器运算过程中有一个错误产生,这将通过使"协处理器错"($\overline{\text{ERROR}}$)输入信号变为逻辑 0 来通知 80386 DX。

3.5.2 微处理器的总线时序

为了实现微处理器与存储器或 I/O 接口的连接与通信,必须了解总线上有关信号的时间关系。这就是本节所要讨论的微处理器的总线时序问题。总线时序是微处理器功能特性的一个重要方面。

1. 总线时序基本概念

1) 指令周期、总线周期及时钟周期

如前所述,指令的执行通常由取指令、译码和执行等操作步骤组成,执行一条指令所需要的时间称为指令周期。不同指令的指令周期是不相同的。

CPU 与存储器或 I/O 接口交换信息是通过总线进行的。CPU 通过总线完成一次访问存储器或 I/O 接口操作所需要的时间,称为总线周期。一个指令周期由一个或几个总线周期构成。

指令的执行是在时钟脉冲(CLK)的统一控制下一步一步地完成的,时钟脉冲的重复周期称为时钟周期(clock cycle)。时钟周期是 CPU 执行指令的基本时间计量单位,它由计算机的主频决定。例如,8086 的主频为 5MHz,则一个时钟周期为 200ns; Pentium Ⅲ的主频为 500MHz,则其时钟周期仅为 2ns。时钟周期也称 T 状态(T-State)。

对于不同型号的微处理器,一个总线周期所包含的时钟周期数并不相同。例如,8086 的一个总线周期通常由 4 个时钟周期组成,分别标以 T_1、T_2、T_3 和 T_4; 而从 80286 开始,CPU 的一个总线周期一般由两个时钟周期构成,分别标以 T_1 和 T_2。

2）等待状态和空闲状态

通过一个总线周期完成一次数据传送，一般要有输出地址和传送数据两个基本过程。例如，对于由 4 个时钟周期构成一个总线周期的 8086 来说，在第一个时钟周期（T_1）期间由 CPU 输出地址，在随后的 3 个时钟周期（T_2、T_3 和 T_4）用来传送数据。也就是说，数据传送必须在 T_2～T_4 这 3 个时钟周期内完成。否则，由于在 T_4 周期之后将开始下一个总线周期而会造成总线操作的错误。

在实际应用中，当一些慢速设备在 T_2、T_3、T_4 三个时钟周期内不能完成数据读写时，那么总线就不能被系统所正确使用。为此，允许在总线周期中插入用以延长总线周期的 T 状态，称为插入“等待状态”（T_w）。这样，当被访问的存储器或 I/O 接口无法在 3 个时钟周期内完成数据读写时，就由其发出请求延长总线周期的信号到 CPU 的 READY 引脚，8086 CPU 收到该请求信号后就在 T_3 和 T_4 之间插入一个等待状态 T_w，插入 T_w 的个数与发来请求信号的持续时间长短有关。T_w 的周期与普遍 T 状态的时间相同。

另外，如果在一个总线周期后不立即执行下一个总线周期，即总线上无数据传输操作，此时总线则处于所谓“空闲状态”，在这期间，CPU 执行空闲周期 T_i，T_i 也以时钟周期 T 为单位。两个总线周期之间出现的 T_i 的个数随 CPU 执行指令的不同而有所不同。

图 3.16 表示了 8086 CPU 的总线周期及其“等待状态”和“空闲状态”的情况。

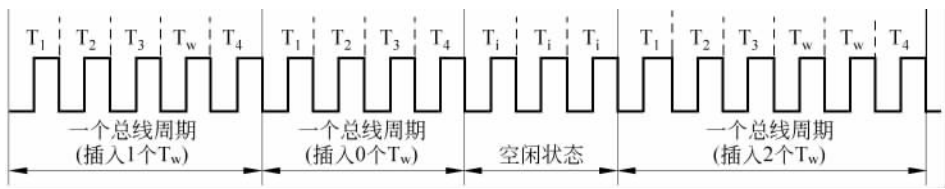


图 3.16　总线时序中的等待状态及空闲状态

3）非流水线和流水线总线周期

有两种不同类型的总线周期：“非流水线总线周期”和“流水线总线周期”。下面讨论这两种总线周期的特点和不同。

采用“非流水线总线周期”，不存在前一个总线周期的操作尚未完成即预先启动后一个总线周期操作的现象，即不会产生前后两个总线周期的操作重叠（并行）运行的情况。图 3.17 表示了一个典型的“非流水线总线周期”时序，注意图中的一个总线周期是由两个时钟周期构成的。

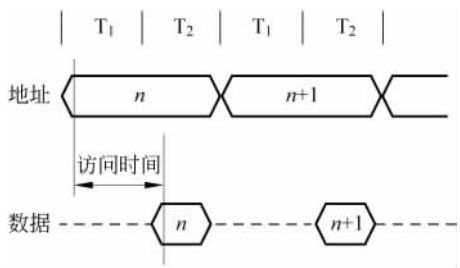


图 3.17　典型的“非流水线总线周期”时序

由图 3.17 可见，在总线周期的 T_1 期间，CPU 在地址总线上输出被访问的存储单元（或 I/O 端口）的地址、总线周期指示码及有关的控制信号（图中仅画出了地址信号，其他信号省略未画），在写周期的情况下，被写数据也在 T_1 期间输出在数据总线上；在总线周期的 T_2 期间，数据被写入所选中的存储单元或 I/O 端口（写总线周期），或把从存储单元或 I/O 端口读出的数据稳定地放置在数据总线上（读总线周期）。

在图 3.17 中可以看到，整个事件序列起始于 T_1 状态的开始时刻，此时第 n 个总线周期的地址码输出在地址总线上。在该总线周期的后继时间，地址总线上的地址仍然有效，而读/写的数据则传送在数据总线上。注意，图中对第 n 个总线周期的数据传送是在该总线周期的 T_2 状态完成的。此时，并未开始输出下一个总线周期的地址信息。另外，图中标出的“访问时间”是反映总线操作速度的一个重要参数，它是指从地址信号稳定地出现在地址总线上到实际发生数据读/写的总的时间。

下面看一下“流水线”式的微处理器总线周期的情形。前面介绍微处理器引脚 NA 时已经提及，所谓“流水线总线周期”，是指对后一个总线周期的寻址与前一个总线周期的数据传送相重叠。也就是说，对后一个总线周期的地址、总线周期指示码及有关的控制信号输出于前一个总线周期的数据传送期间。图 3.18 给出了一个流水线总线周期的典型时序。

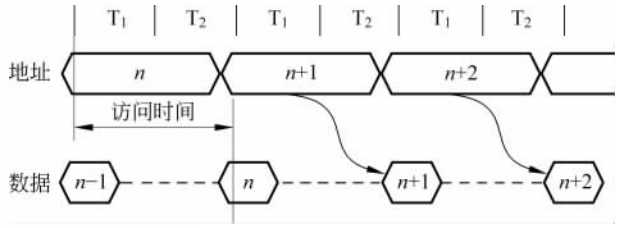


图 3.18　流水线总线周期时序

由图 3.18 可见，第 n 个总线周期的地址在该总线周期的 T_1 开始时刻变为有效，然而该总线周期的数据却出现于第 $n+1$ 个总线周期的 T_1 状态；而在第 n 个总线周期的数据传送的同时，第 $n+1$ 个总线周期的地址便输出到地址总线上了。由此可以看到，在流水线总线周期中，当微处理器进行前一个已寻址存储单元的数据读/写的同时，即已开始了对后一个被访问存储单元的寻址。或者说，当第 n 个总线周期正在进行之时，第 $n+1$ 总线周期就被启动了。从而使前后两个总线周期的操作在一定程度上得以并行进行，这样可以在总体上改善总线的性能。

前面已经介绍，可通过插入等待状态来扩展总线周期的持续时间。这实际上是通过检测$\overline{\text{READY}}$输入信号的逻辑电平来实现的。$\overline{\text{READY}}$输入信号也正是为此目的而提供的。该输入信号在每个总线周期的结尾时刻被采样，以确定当前的总线周期是否可以结束。如图 3.19 所示，在$\overline{\text{READY}}$输入端上的逻辑 1 电平表示当前的总线周期不能结束。只要该输入端保持在逻辑 1 电平，说明存储器或 I/O 设备的读/写操作还未完成，此时应将当前的 T_2 状态变成等待状态 T_w 以扩展总线周期。直到外部硬件电路使$\overline{\text{READY}}$回到逻辑 0 电平，这个总线周期才能结束。具体地说，在每个总线周期的结尾时刻（T_2 结束

时)对$\overline{\text{READY}}$信号进行采样,以确定当前的“时钟周期”是 T_2 还是 T_w。如果这时$\overline{\text{READY}}=0$,表明当前总线周期可以结束,即当前时钟周期为 T_2;如果$\overline{\text{READY}}=1$,则当前时钟周期为 T_w,并且微处理器将继续检测$\overline{\text{READY}}$直到其为 0,总线周期才能结束。这种扩展总线周期的能力允许在较高速的微型计算机系统中可以使用较低速的存储器或 I/O 设备。

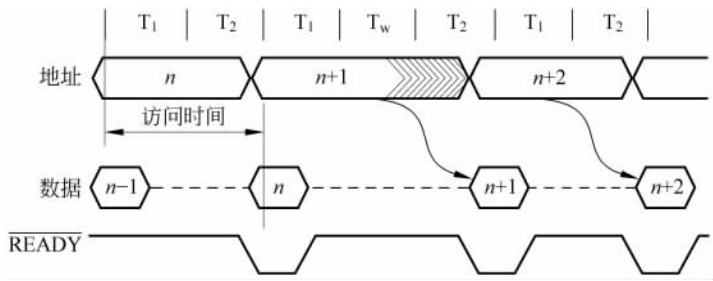


图 3.19 带等待状态的流水线总线周期时序

2. 基本的总线时序

为了对总线操作过程有一个基本的了解,先让我们看一下经适当简化的 8086 写、读总线周期时序。在此基础上,可进一步了解较为复杂的总线操作时序。

我们已经知道,微处理器通过 3 种总线(地址总线、数据总线和控制总线)与存储器或 I/O 接口进行连接与通信。为了把数据写入存储器(或 I/O 接口),微处理器首先要把欲写入数据的存储单元的地址输出到地址总线上,然后把要写入存储器的数据放在数据总线上,同时发出一个写命令信号($\overline{\text{WR}}$)给存储器。

一个简化的 8086 写总线周期时序如图 3.20 所示。其中,请注意两点:第一,8086 的一个总线周期包含 4 个时钟周期(即 T_1、T_2、T_3 和 T_4);第二,8086 采用地址和数据总线复用技术,即在一组复用的“地址/数据”总线上,先传送地址信息(T_1 期间),然后传送数据信息(T_2、T_3、T_4 期间),从而可以节省微处理器引脚。图中的 M/$\overline{\text{IO}}$是 8086 的输出信号,用以表明本总线周期是访问存储器还是访问 I/O 接口。具体而言,M/$\overline{\text{IO}}=1$,是访问存储器;M/$\overline{\text{IO}}=0$,则为访问 I/O 接口。

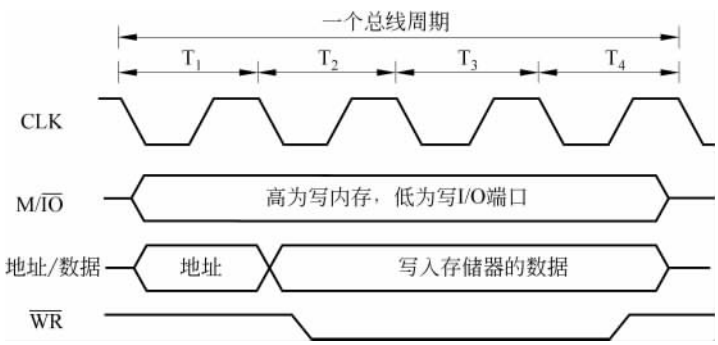


图 3.20 简化的 8086 写总线周期时序

若要从存储器读出数据，则微处理器首先在地址总线上输出所读存储单元的地址，接着发出一个读命令信号($\overline{RD}$)给存储器，经过一定时间(时间的长短决定于存储器的工作速度)，数据被读出到数据总线上，然后微处理器通过数据总线将数据接收到它的内部寄存器中。一个简化的 8086 读总线周期时序如图 3.21 所示。

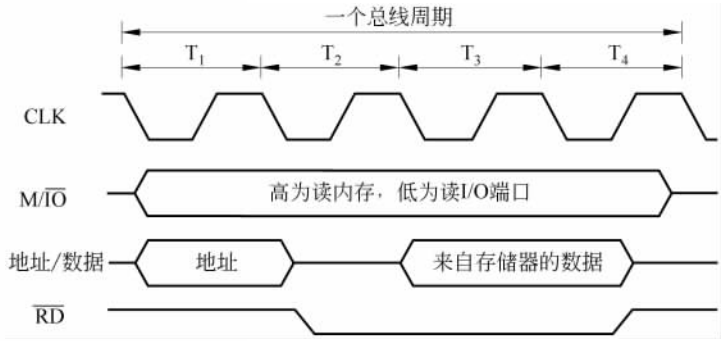


图 3.21　简化的 8086 读总线周期时序

另外，正如前面在介绍“等待状态”的概念时所提到的，若被访问的存储器或 I/O 接口的工作速度较慢，不能在预定的时间完成数据读/写操作，则可通过在总线时序中插入等待状态(T_w)来扩展总线周期；而是否插入 T_w，可通过检测 8086 的$\overline{READY}$输入引脚的逻辑电平来决定。关于这方面的情况，在简化的读/写总线周期时序图中省略未画。

习题 3

3.1　80386 以上的微处理器通常有哪几种工作模式？各自的主要特点是什么？

3.2　简要说明 80x86/Pentium 处理器编程结构中所包含寄存器的主要类型及寄存器名称。

3.3　8086/8088 CPU 标志寄存器中有哪几个状态标志位和控制标志位？它们各自的功能是什么？

3.4　为什么要将存储系统空间划分成许多逻辑段，分段后如何寻址要访问的存储单元？

3.5　什么是物理地址？什么是逻辑地址？物理地址与逻辑地址有何联系？

3.6　什么是段基值？什么是偏移量？如何根据段基值和偏移量计算存储单元的物理地址？

3.7　在 80x86 实模式下，若 CS＝1200H，IP＝0345H，则物理地址是什么？若 CS＝1110H，IP＝1245H，则物理地址又是什么？

3.8　某存储单元的物理地址为 28AB0H，若偏移量为 1000H，则段基值为多少？

3.9　若 80x86 实模式下当前段寄存器的基值 CS＝2010H，DS＝3010H，则对应的代码段及数据段在存储空间中物理地址的首地址及末地址是什么？

3.10　设现行数据段位于存储器 10000H～1FFFFH 单元，则 DS 寄存器的内容应为

多少？

3.11 什么是堆栈？它有什么用途？堆栈指针的作用是什么？举例说明堆栈的操作。

3.12 在80x86实模式系统中，堆栈的位置如何确立？由SS寄存器的值所指定地址的位置是不是栈底？为什么？

3.13 某系统中已知当前SS＝2100H，SP＝080AH，说明该堆栈段在存储器中的物理地址范围。若在当前堆栈中存入10个字节数据后，那么SP的内容变为何值？

3.14 简要说明Pentium处理器内部所包含的主要功能部件。

3.15 在片内Cache的设置上，Pentium与80486有何不同？

3.16 以Pentium处理器为例，解释现代微处理器设计中所采用的下列技术：流水线方式；流水级；超级流水线（超流水）；超标量结构。

3.17 简要说明Pentium处理器实现“动态转移预测”的基本方法及工作过程。

3.18 80386 DX CPU的外部引脚信号共分哪几类？对于一个引脚信号，通常从哪几个方面对其进行描述？试举两例。

3.19 当80386 DX输出的字节允许信号$\overline{BE_3}$ $\overline{BE_2}$ $\overline{BE_1}$ $\overline{BE_0}$＝0000时，将产生哪种类型的数据传送（字节、字、双字）？数据传送将通过哪些数据线进行？

3.20 说明“非流水线总线周期”和“流水线总线周期”的各自特点。

3.21 微机A和微机B采用主频不同的CPU芯片，在片内逻辑电路完全相同的情况下，若A机CPU的主频为8MHz，B机为12MHz，且已知每台机器的总线周期平均含有4个时钟周期，A机的平均指令执行速度为0.4MIPS，那么该机的平均指令周期为多少微秒（μs）？每个指令周期含有几个总线周期？B机的平均指令执行速度为多少MIPS？

注：MIPS（Million Instruction Per Second，每秒百万条指令），描述计算机执行指令速度的指标，每秒执行一百万（10^6）条指令，其指令执行速度为1MIPS。