第二章　微处理器的基础知识

第二章　微处理器的基础知识

- 2.1 计算机系统的基本结构
- 2.2 计算机系统的基本工作原理
- 2.3 机器指令
- 2.4 8086CPU内部结构
- 2.5 80x86 系列PC机的内部存储器组织
- 2.6 80x86 系列PC机的I/O端口组织

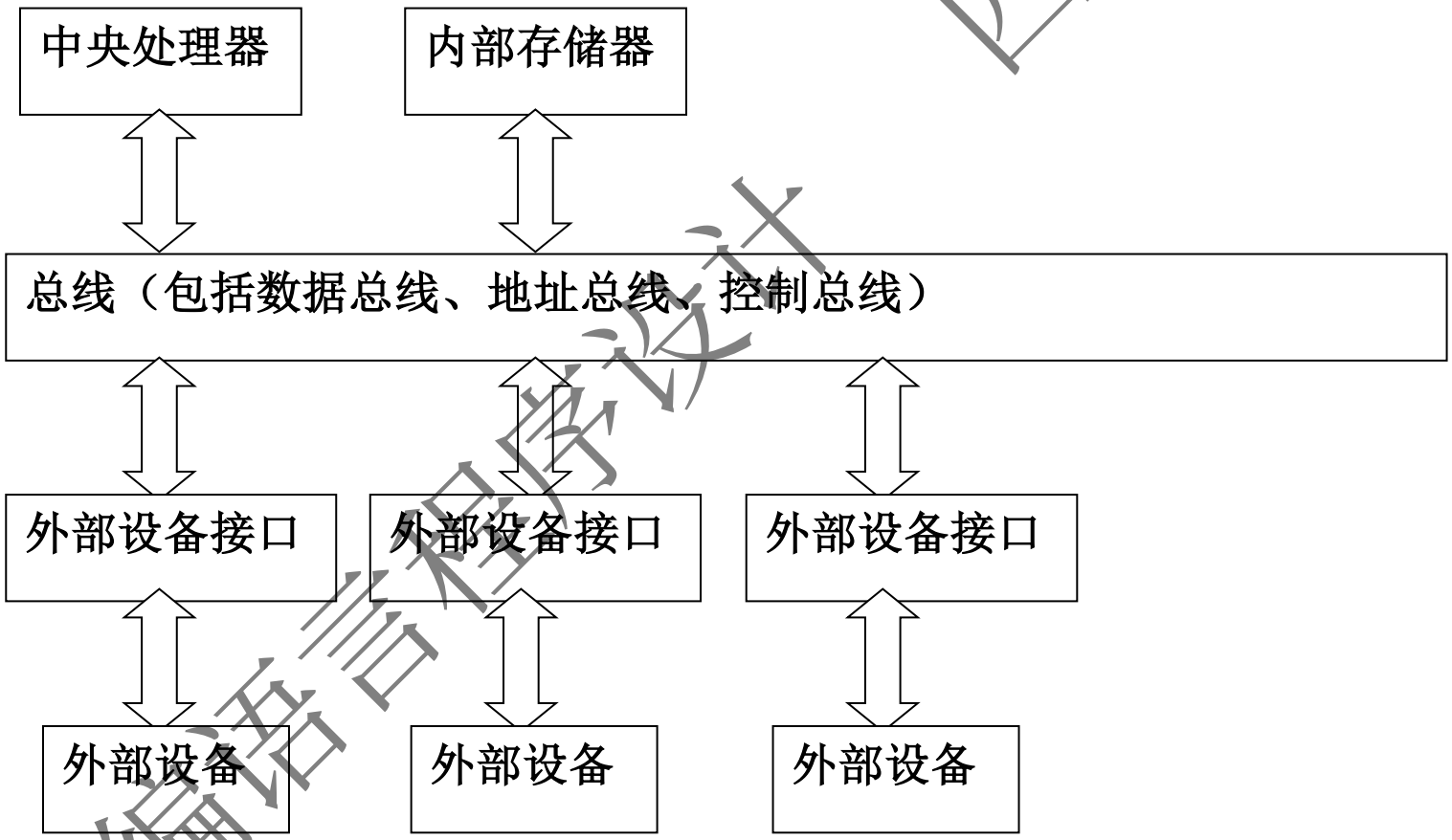
2.1 计算机硬件系统的基本结构

- 汇编语言是面向硬件的语言，是和计算机硬件系统的工作机制密不可分的。
- 不理解硬件系统的工作原理，则无法掌握汇编语言。
- 不结合硬件机制理解汇编语言，则对硬件系统无法深入理解。

2.1 计算机硬件系统的基本结构

- 计算机系统结构是指**硬件结构**，主要包括中央处理器（CPU）、总线（BUS）、内部存储器（memory）、外部设备接口（interface）、外部设备（device）这5种功能部件。

2.1 计算机硬件系统的基本结构



2.1 计算机硬件系统的基本结构

- 核心概念

- 1.存储单元：**寄存器**、**内存单元**、**端口**

- 2.**地址**：存储单元的编号

中央处理器（CPU）

- （1）中央处理器（CPU）：计算机系统的中央处理部件，控制整个计算机系统的运转过程，其功能相当于人类的大脑。

- 主要功能：

- (a)自动完成各类时序过程；自动从内存单元中**读取、执行机器指令是其中一种**；

中央处理器（CPU）

- (b)按照指令的功能要求，对存储单元进行读写操作；
- (c)按照指令的功能要求，使用ALU执行算术、逻辑运算；
- (d)提供一定数量的寄存器，对正在处理的数据进行暂时存储，或者存放逻辑地址，例如下一条指令的地址。

中央处理器（CPU）

Pentium 4

Pentium

80386

8088CPU

总线（BUS）

- （2）总线（BUS）：连接计算机系统中其他功能部件的桥梁，是计算机系统中信号传递的枢纽。

- 只有通过总线，计算机系统的各部件才能实现相互通信。

数据总线

- 主要功能：
- (a)传输数据：各功能部件通过总线可以进行数据交换。
- 用于完成数据传输功能的这组总线称为数据总线。

地址总线

- (b)发送地址：

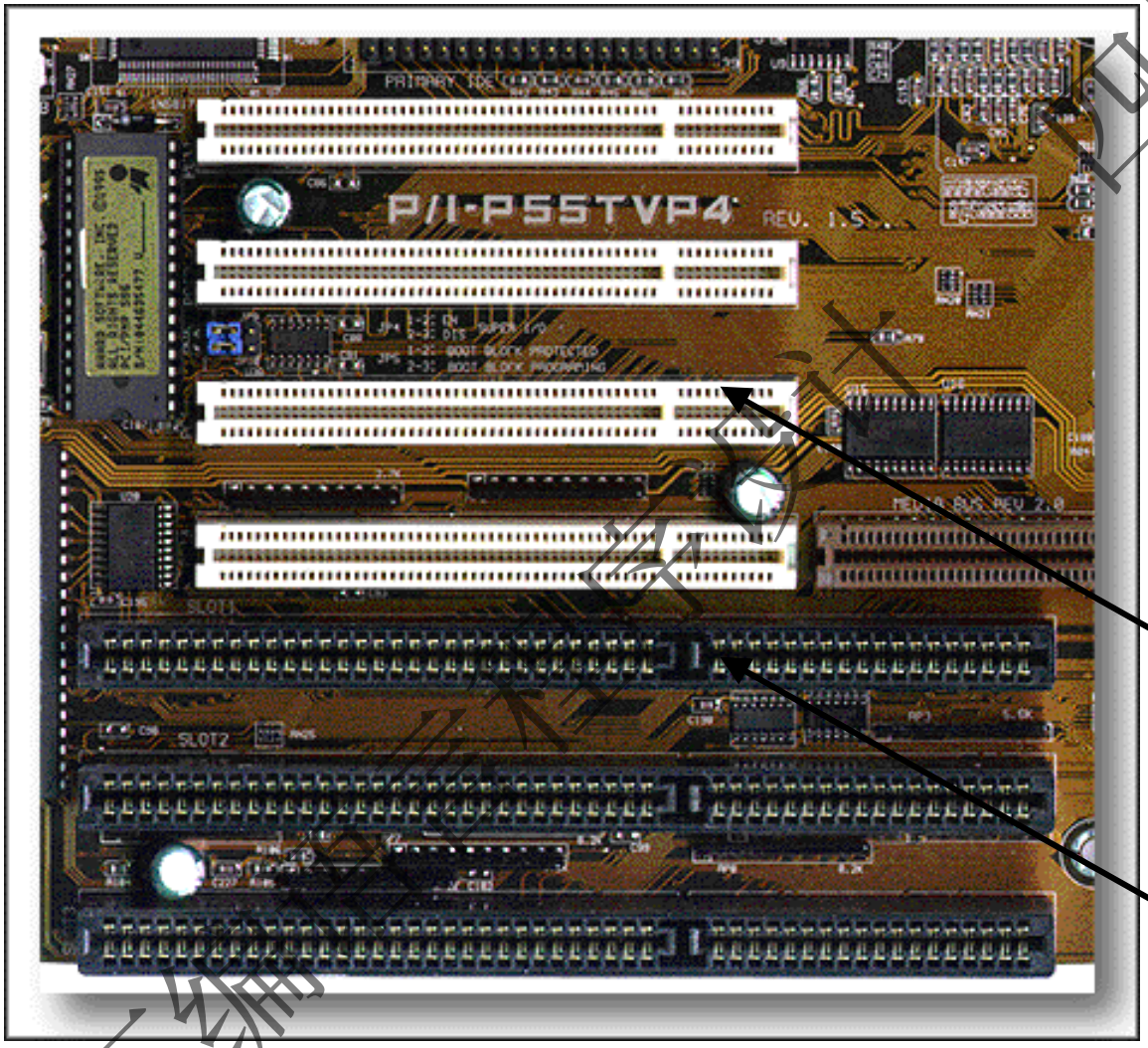
- 计算机系统中的各功能部件中都有若干存储单元，进行数据传输时必须指明存储单元位置，即地址。

- 用于完成地址发送功能的这组总线称为地址总线。

控制总线

- (c)传送控制信号和状态信号：
- 通过控制信号的发送，CPU能够控制其他功能部件完成所指定的操作（例如读、写操作）；
- 其他功能部件通过向CPU发出自己的状态信号，向CPU回馈自己的工作状态（例如中断请求）。
- 用于传送控制信号和状态信号的这组总线称为控制总线。

总线（BUS）



内部存储器（memory）

- （3）**内部存储器**（memory）：计算机系统的内部存储部件，用于存放当前运行的程序和程序所使用的数据，是计算机系统的存储中心。

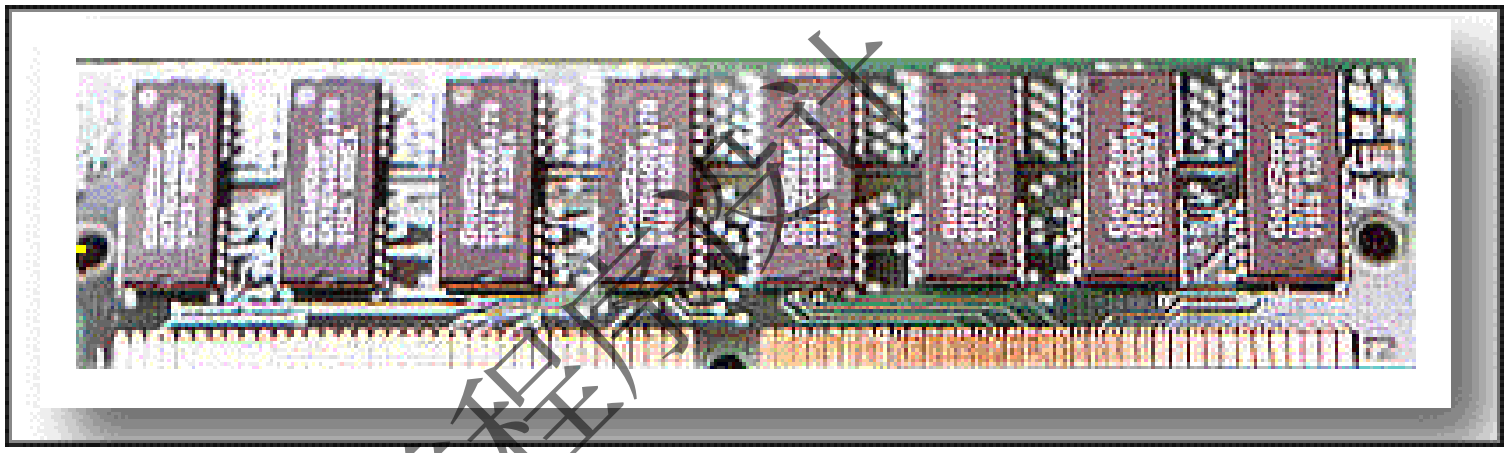
- 任何程序如果要取得CPU控制权并得到运行，必须先被装入内存。

内部存储器（memory）

- 内部存储器能够通过**地址总线**上传送来的地址和**控制总线**上发来的控制信号判断CPU**是否选中**自己进行操作。

- 通过控制总线上发来的控制信号决定是将数据总线上的数据写入内存单元（**写数据**），还是将内存单元中的数据放到数据总线上（**读数据**）。

内部存储器（memory）



外部设备接口（interface）

- （4）外部设备接口（interface）：接口是计算机系统中用于**连接外部设备和总线的中间电路**。

- 只有通过接口，外部设备才能与总线建立连接。

接口存在的必要性

- 外部设备接口所提供的主要功能如下：
- (a)**信号转换**
- **外部设备**在功能、形式、处理速度、信号标准等多方面**千差万别**。

接口存在的必要性

- **系统总线则是按照严格的工业标准设计的**，对信号名称、信号含义、信号形式、信号传送速度等方面都有严格规定。

- 外部设备的生产厂家必须提供相应的接口电路来完成外部设备信号和总线信号之间的相互转换；

接口存在的必要性

- (b)协调速度差异

- 为CPU处理速度与外部设备处理速度之间的差异提供缓冲，避免数据丢失。

接口的主要功能

- (c)**提供CPU与外部设备交互的途径**：
- 可以接收CPU发来的**控制信号**，也可以向CPU发送**状态信号**；
- 通过总线，接口**可以和CPU交换数据（数据字节）**。
- 可以从CPU接收**命令字节**并完成该命令对应的功能，可以向CPU提供**状态字节**满足CPU的查询需求。

接口的主要功能

- (d)可编程功能：
- 接口中提供一定数量的存储单元，分别用于存放输入或输出数据、CPU发来的命令信息、为CPU提供的状态信息等。
- 这些存储单元由于位于接口中，所以被称为端口，分为数据端口、状态端口、命令端口。
- CPU可以使用指令访问端口，从而完成CPU与接口的信息交换。

接口的可编程功能

- 如果把这些**访问端口的指令组织起来**，编制一个完整程序来控制接口，使它正确完成对外部设备的数据输入输出功能，这样的程序称为**接口的驱动程序**。

- 一般**操作系统中**都为程序设计人员提供各类硬件接口的**系统调用**，这些系统调用就是通过**驱动程序**来完成的。

接口实例

- 举例来说：
- 显示器连接到总线上需要显卡
- 音箱连接到总线上需要声卡
- 键盘连接到总线上需要键盘接口

外部设备（I/O device）

- （5）外部设备（I/O device）：

- 也称**I/O设备**，是计算机系统**与外部世界交互所必须的**功能部件，同时也是**人机交互的必要工具**。

- 外部设备的形式、功能与其具体的I/O应用相关，由于外部信息种类很多，所以相应设备的**形式和功能也多样化，很难统一**。

2.2计算机系统的基本工作原理

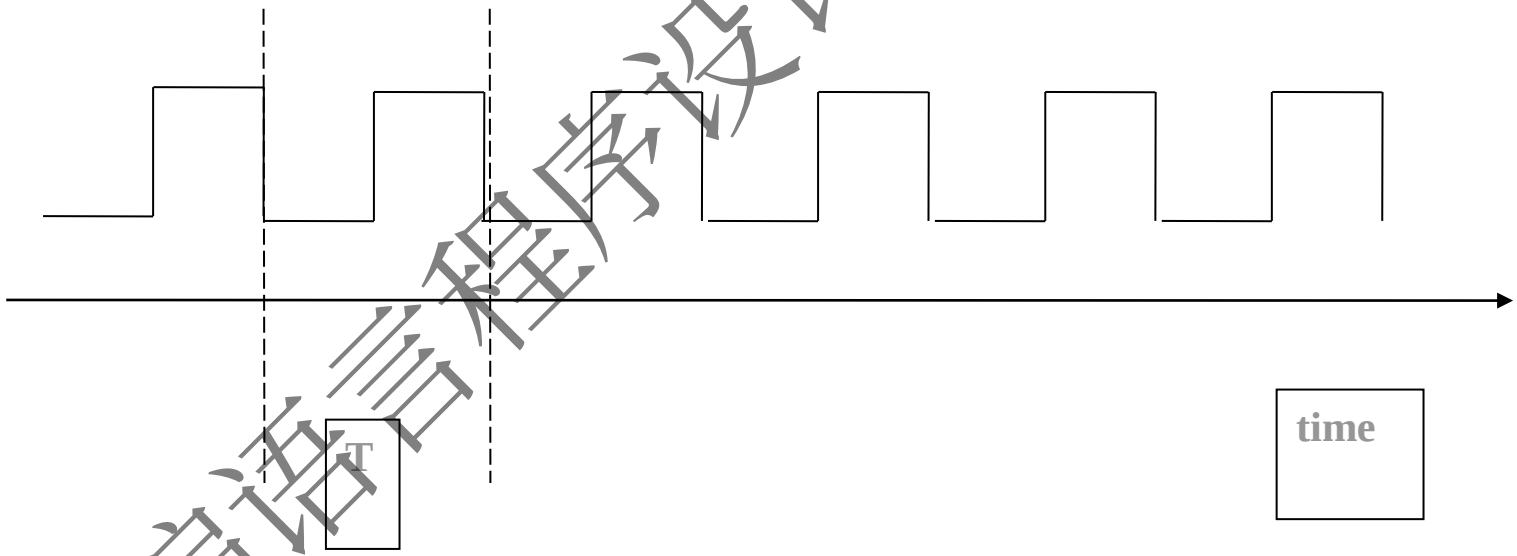
- （1）计算机系统中的基本信号

- (a)**时钟信号**（clock）：这是一个周期性的方波信号，由时钟发生器产生，并发送给CPU。

- 时钟信号的每一个周期称为一个**时钟周期（节拍）**，节拍是计算机系统中最小的时间单位。

时钟信号

- 时钟信号的频率与系统的主频成正比（主频即晶振信号的频率，晶振信号用于分频产生时钟信号）。



时钟信号

- **时钟信号**的功能相当于**人的脉搏**，是**最基础的控制信号**，是**时序控制的基础**。

- 任何其它控制信号维持的时间都是节拍的整数倍。

- CPU控制其它功能部件以时钟信号为**时间基准**进行时序操作。

时钟信号

- **节拍的长短**是衡量计算机系统**运转速度**的一个**关键指标**，节拍越短说明主频越高，系统的运行速度也越快。

- 例：A与B两个系统从内存中读取一个字节数据都需要6个节拍，但A的节拍比B短，A系统原则上就比B系统快。

读信号和写信号

- (b) 读信号(read)和写信号(write)：CPU对其它功能部件实施的操作都可以概括为数据的读和写。

- 如果CPU要对CPU以外的存储单元进行读写操作，则通过控制总线发出读写信号。

中断信号和中断响应信号

- (c) **中断（请求）信号**(interrupt)和中断响应信号(interrupt accept)：

- 中断信号是一种状态信号，由外设接口通过控制总线向CPU发出。

- 中断响应信号是一种控制信号，由CPU通过控制总线向外设接口发出。

中断信号和中断响应信号

- **发出中断信号的原因**：
- 输入设备接收到输入数据，请求CPU读取输入数据；
- 输出设备的输出数据已经输出完毕，请求CPU发送新的输出数据；
- 设备出现了故障。

中断信号和中断响应信号

- 中断信号要求CPU暂时**中断正在执行的程序**，调用中断服务程序处理外设接口提出的请求。
- 在接收到中断信号后，CPU**根据当前情况决定是否响应**。
- 若CPU不响应，则接口保持中断信号，直到被响应时，才从总线上撤消中断信号。
- 响应中断请求并执行中断服务程序后CPU**继续执行被中断程序**。

计算机系统的时序过程

- 计算机系统的时序过程：
- 计算机系统**以节拍为基本单位**、**按照既定的顺序**产生控制信号，从而**完成一系列信号传递的过程**。

计算机系统的时序过程

- 计算机系统的**时序过程**，分**两个方面**：
- （1）**执行指令**：CPU**以节拍为单位**、**按顺序、分步骤执行**指令所要求的操作；
- （2）**其他时序过程**：
- 1）CPU自动从内存单元读取下一条指令
- 2）CPU**接收**外部设备接口发来的**请求信号**，并向接口**发送控制信号**来完成对外部设备的控制。

计算机系统的时序过程

- **时序过程示例（读内存单元，每个步骤占用一个节拍）：**

- 1）T1节拍： CPU发送存储单元地址到地址总线，地址译码开始

- 2）T2节拍：地址总线上信号已稳定，CPU发送读信号到控制总线

计算机系统的时序过程

- T3节拍：读信号、地址信号均稳定的加在所有连接在总线的部件上，已选通要操作的内存芯片。内存单元读操作开始。

- T4节拍：从内存单元读取的数据在数据总线上已稳定，CPU从总线上读取数据

计算机系统的时序过程

- 内存单元、端口的读写过程占用的时间片段称为**读写周期**，是节拍的整数倍。

- 读和写是最基本的时序过程，理解了这两个过程，其它时序过程可以按照相同的原理逐步理解。

2.3 机器指令

- **机器指令**是**一种二进制编码**，编码**与指令功能一一对应**。

- 每一种计算机都有自己的一套**指令系统**，每条指令代表一种固定的功能。

执行机器指令的时序过程

- （1）机器指令经过CPU的指令译码器**译码**，转换为各种**控制信号**

- （2）控制信号经过CPU中逻辑电路和时序电路的处理，和时钟信号结合后**按照先后顺序**向控制总线发送

- （3）CPU通过按时序产生的控制信号控制计算机系统其它部件的动作。

机器指令与系统硬件的关系

- 每种计算机的指令系统可能各不相同，指令译码器也各不相同。
- 机器指令总是和某种具体的计算机系统结合在一起的，没有脱离具体硬件的机器指令。

指令周期

- 执行一条指令所占用的时间片段称为**指令周期**，它是**节拍的整数倍**。
- 指令中经常出现的操作是读写操作，所以**指令周期内经常包含读写周期**。
- 用于在CPU内部完成数据运算的时间片段称为**运算周期**。

机器指令与时序过程的关系

- 计算机系统中的时序过程比指令的级别更底层，**读写周期不一定包含在指令周期内**。
- 除执行指令外，CPU常常使用其他时序过程来完成对硬件的控制。

指令的原子性

- 指令具有**原子性**，虽然指令周期由多个节拍构成，但是一条指令的执行过程**不允许被打断**。(DMA操作除外)

- 即便是响应中断信号，也必须等到当前指令执行完毕后CPU才会去响应。

指令的原子性

- 一条指令的功能要么被CPU完全执行，要么完全不执行，不会出现只执行半条指令的情况，除非出现了严重的硬件故障。

第二章　微处理器的基础知识

2.4 Intel 80x86系列CPU内部结构

- **汇编语言建立在机器指令基础之上，是一种描述硬件运转过程的语言。**

- 每一种CPU都有自己的指令系统，因此在理解汇编语言基本概念前，必须先对计算机系统、CPU的内部结构和工作机制作一个概要的理解。

本课程选用的CPU类型

- Intel 推出的80x86系列处理器的性能和功能越来越强。
- 但是，从汇编语言程序角度看，8086建立的实模式和80386建立的保护模式模型到目前为止一直适用。
- 本门课程的主要内容：以8086为例说明实模式编程。

Intel80x86系列微处理器性能对比

CPU	数据总线宽度	地址总线宽度	寻址能力	工作模式
8086	16	20	1MB字节	实模式
8088	8	20	1MB字节	实模式
80286	16	24	16MB字节	实模式、保护模式
80386SX	16	24	16MB字节	实模式、保护模式
80386DX	32	32	4GB字节	实模式、保护模式
80486DX	32	32	4GB字节	实模式、保护模式
Pentium	64	36	64GB	实模式、保护模式

8086/8088 CPU内部结构

- 8086和8088CPU内部结构是一致的，内部处理的最长二进制编码都是16位的。

- 但是8086针对的数据总线是16位，8088是8位，两种CPU只是外部引脚有所区别。

执行指令与读取指令

- **读取指令**：从地址总线发出内存地址，将内存中的指令逐条读取到CPU内部。
- **执行指令**：对机器指令译码，按时序产生控制信号，从而完成指令规定的时序过程。
- 两种操作都是**CPU自动进行**的。
- 程序能够控制CPU什么时候取指令、执行指令吗？

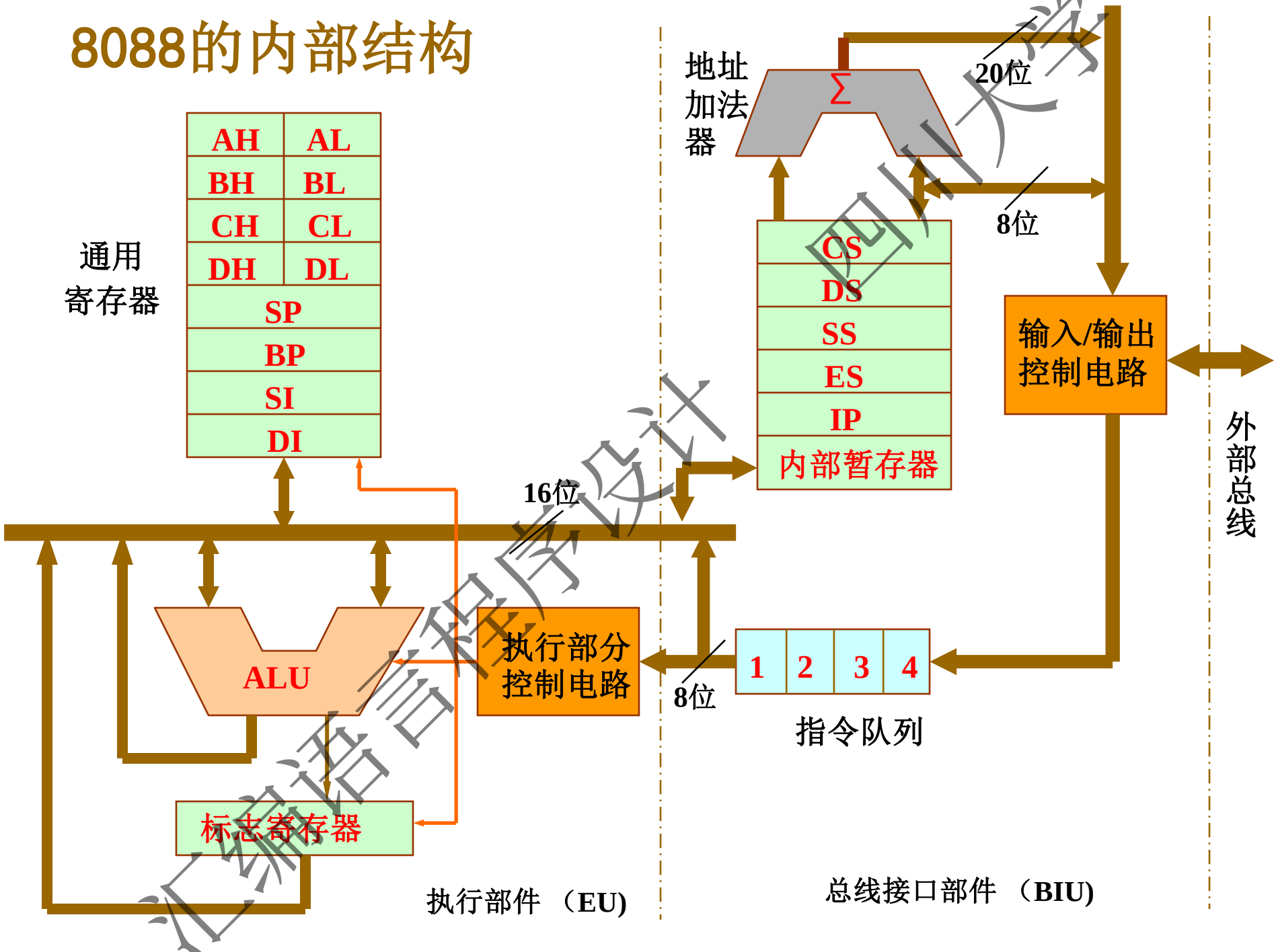
执行指令与读取指令

- 读取指令：一定使用总线
- 执行指令：不一定每个节拍都使用总线
- 考虑**流水线机制**：执行当前指令时，如果不使用总线，则同时读取下一条指令。

8086/8088 CPU内部结构

- **EU**（Execute Unit）：执行部件
- **BIU**（Bus Interface Unit）：总线接口部件。
- 有时它们在时间上进行交叠，形成了流水线工作方式，加快了CPU运转速度

8088的内部结构



8086/8088 CPU内部结构

- EU中的核心部件：

- (1) **控制器**：计算机系统中控制所有功能部件（包括控制器本身）协同工作，**自动执行计算机时序过程（包括程序）的功能部件**。

控制器执行指令的原理

- 1)通过对机器指令译码产生控制信号
- 2)按照指令规定的时序发出控制信号
- 3)产生一系列时序过程来完成指令功能

8086/8088 CPU内部结构

- (2) **运算器**：计算机系统中**加工、处理数据的功能部件**，其功能包括算术运算和逻辑运算。

8086/8088 CPU内部结构

BIU中的核心部件：

（1）**总线控制逻辑**：CPU与总线接口的逻辑电路，CPU与控制、地址、数据总线交互都必须使用这一功能部件。

8086/8088 CPU内部结构

- （2）**指令队列**：这是一组CPU内部的存储单元，它的功能是存放后续将要执行的指令序列。

- 由于EU和BIU两大模块的分工合作，CPU可以利用当前指令的运算周期启动BIU，将后续指令预先读取到指令队列中。

指令队列

- 当这些后续指令被执行时，CPU就不需要再使用总线到外部的存储单元去读取它们。
- 因为从指令队列读取指令比从外部存储单元读取指令的速度快得多，所以CPU的工作效率得到提高。
- **指令队列对于指令是不可见的**。

8086/8088 CPU内部结构

- (3) **地址加法器**：是CPU用于产生外部存储单元物理地址的器件。

- 由于CPU外部的地址总线是20位的，而CPU内部的寄存器最长为16位，为了协调此差异，CPU把外部存储单元的地址分为段基值和偏移量两个16位分量.

地址加法器

- **物理地址**：CPU通过地址总线定位某个外部存储单元时使用的地址，即真实的存储单元地址，内存物理地址具有20个bit。

- **逻辑地址**：由于CPU内部寄存器的长度限制，在CPU内部使用的存储单元地址形式，包括段基值和偏移量两部分，二者都具有16个bit。

- **地址与内存单元的对应**：无论是物理地址还是逻辑地址，**一个地址对应内存中一个独立的字节单元**。

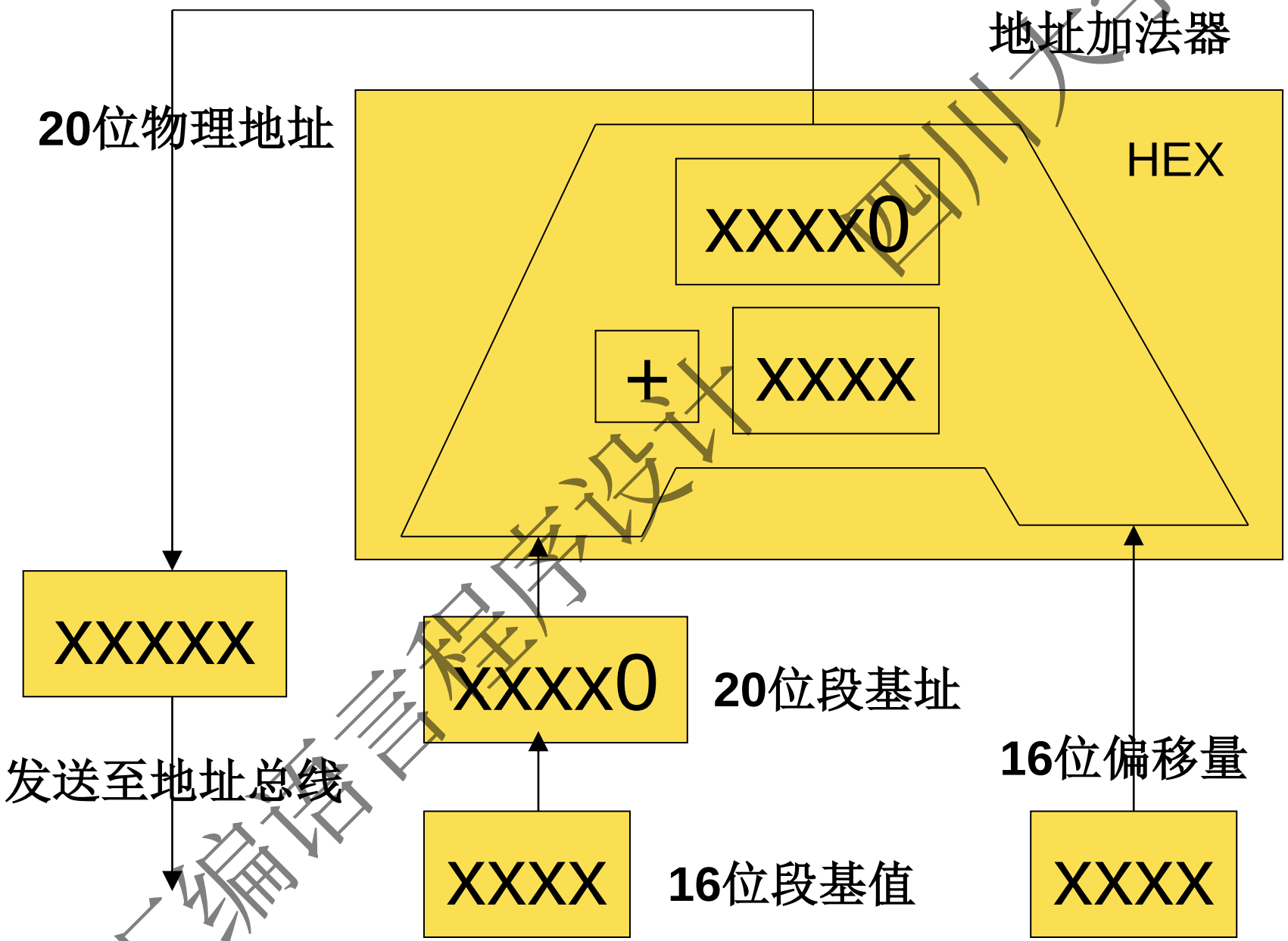
地址加法器

- 段基值：偏移量　表示的地址称**逻辑地址**
- 段基值＊１６＋偏移量＝物理地址（２０位）
- 段基值、偏移量、物理地址都是无符号数（编码）。

物理地址与逻辑地址

- 机器指令中的地址表示形式**仅限于逻辑地址**。

- **任意一个逻辑地址可以转换为唯一物理地址**（反之则不行），转换过程是CPU的BIU单元自动完成的。



8086/8088 CPU内部结构

- **寄存器：CPU内部的少量存储单元**，每个存储单元都能存储二进制数据，并且都有自己独特的功能和相应的名称（地址）。

- 8086/8088 中的寄存器：总共有14个物理寄存器，逻辑上的寄存器有22个。

- 下面就它们的功能分别来讨论。

段寄存器

- （a）　段寄存器：

- 包括CS（Code Segment）、SS（Stack Segment）、DS（Data Segment）、ES（Extra Segment）四个16位物理寄存器。

- 用于存放程序所要使用的4个存储段的**段基值**，分别对应于内存中的四块存储区域，**代码段**、**堆栈段**、**数据段**、**附加段**。

段寄存器

- 每个程序都可能会使用这样四个段，其中代码段是必须的，实用的程序通常至少包含代码段、堆栈段、数据段。

- **段**：内存中一段连续的空间，在程序中具有特定的用途。

- **段基址=段基值*16**　段在内存中的起始地址。

段

- **代码段**：用于存放程序的机器指令序列；
- **堆栈段**：用于存放程序使用堆栈指令所保存的数据；保存子程序调用指令提供的返回地址；保存中断断点；
- 堆栈段的访问一般遵循后进先出（LIFO）原则。

段

- **数据段**：存放程序需要直接使用的数据，包括变量、数组、字符串等；

- **附加段**：内容不确定，可以根据实际需要决定。通常用于存放串操作指令中的目的串。

地址指针寄存器

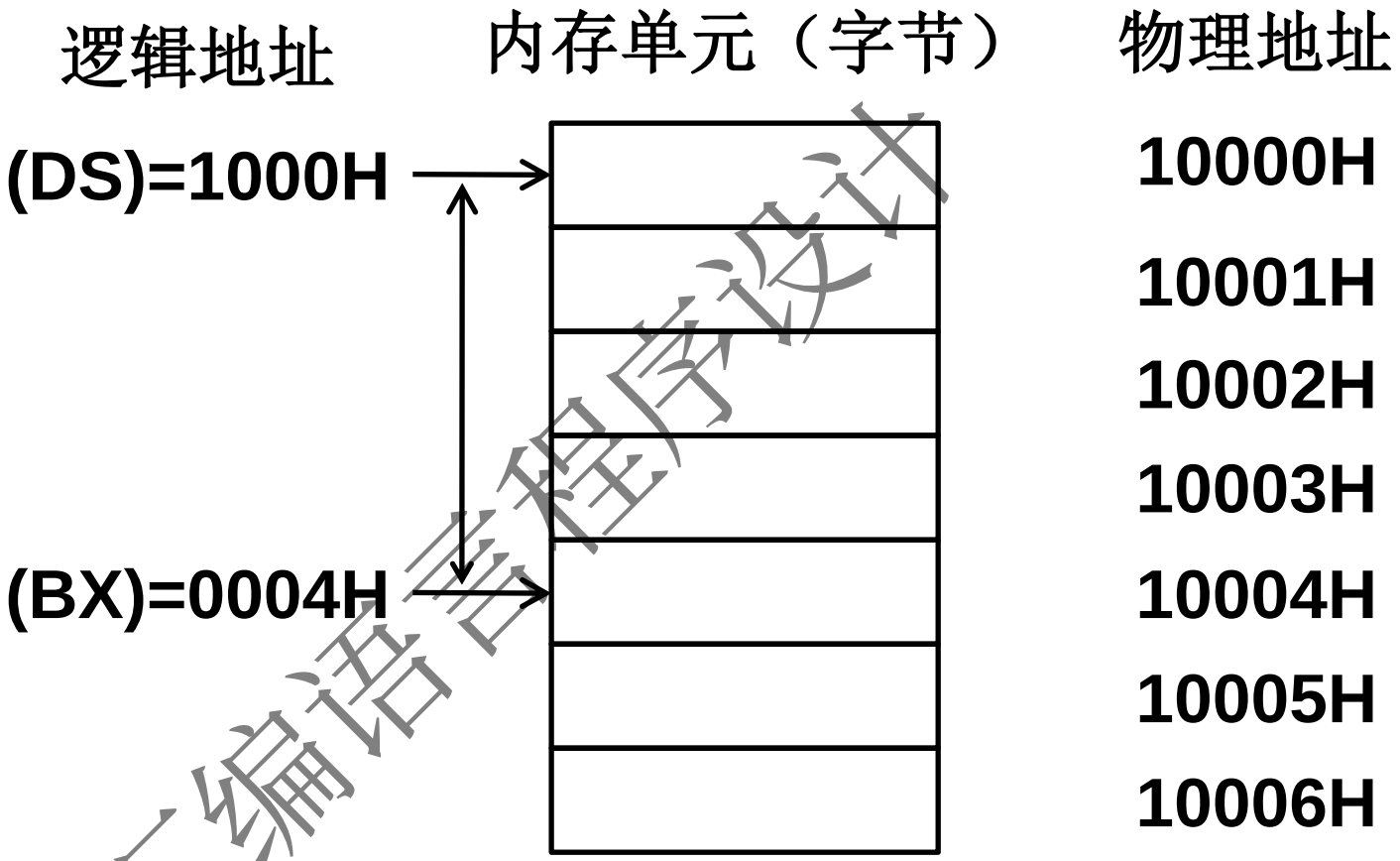
- **（b）地址指针寄存器：**

- 包括BX、SI、DI、BP、SP、IP六个16位寄存器，用于存放逻辑地址的偏移量或者偏移量的一部分（分量）.

- 其作用在寻址时类似于游标，通过相对于段基址的相对**字节距离**来定位具体的字节或字单元。

地址指针寄存器

- 使用**逻辑地址定位内存单元的示例：**



地址指针寄存器

- BX称为**基址寄存器**，可以用于存放偏移量或偏移量分量。

- **通常和DS、ES**这两个段寄存器配合使用，用于定位数据段或附加段中的内存单元；

地址指针寄存器

- SI称为**源变址寄存器**，用于存放偏移量或偏移量分量。

- **通常和DS、ES**这两个段寄存器配合使用，用于定位数据段或附加段中的内存单元。

- 在串操作指令中，SI用于指明源串偏移量，所以被称为源变址寄存器。

地址指针寄存器

- DI称为**目的变址寄存器**，用于存放偏移量或偏移量分量。
- 通常和DS、ES这两个段寄存器配合使用，用于定位数据段或附加段中的内存单元。
- 在串操作指令中，DI用于指明目的串偏移量，所以被称为目的变址寄存器。

地址指针寄存器

- BP称为**基址指针寄存器**，用于存放偏移量，**通常和SS**段寄存器配合使用，用于定位堆栈段中的内存单元。

- SP称为**堆栈指针**，用于存放偏移量，**只能和SS**段寄存器配合使用，且始终指向堆栈的栈顶，在堆栈指令中隐含的使用它来定位栈顶数据。

地址指针寄存器

- IP称为**指令指针**，用于存放偏移量，**只能和CS**段寄存器配合使用，且始终指向代码段中下一条将要读取到CPU指令队列的那条指令；

- 修改IP中内容的操作是CPU在每读取一条指令到指令队列后自动进行的，使它指向要读取的下一条指令。

- **转移指令**可以隐含的修改IP寄存器中的内容。

数据寄存器

- **（c）数据寄存器：**
- 包括**AX、BX、CX、DX**这样4个16位的寄存器.
- 在逻辑上一个16位的数据寄存器可以看成是3个寄存器.
- 例如**AX**，可以把它作为一个16位的数据寄存器来使用，也可以把高低8位分开，高8位为**AH**，低8位为**AL**.

数据寄存器

- 在使用上要注意逻辑上不同的寄存器可能在物理上是相互覆盖的。

- 这里的寄存器BX在上面提到过，它既可以用作数据寄存器，也可以用作地址指针寄存器。

标志寄存器

- **（d）标志寄存器：**

- 8086CPU提供一个16位的标志寄存器FR，对这个寄存器的使用在指令中**往往是隐含的**。

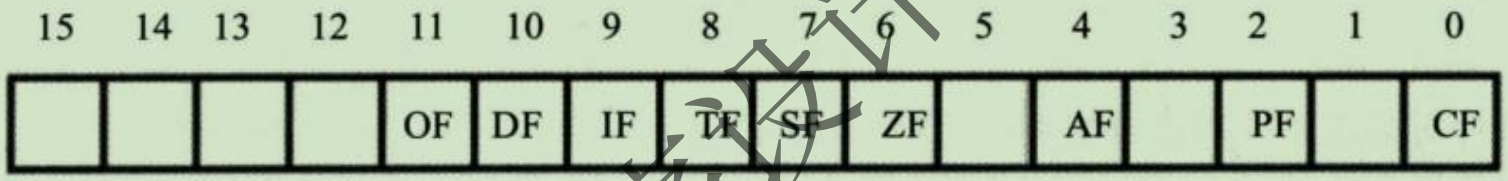
- 值得注意的是，FR是按位操作的，**每一个二进制位都有各自特定的含义**。

标志寄存器

- 一般在汇编语言程序中的运算指令或者标志位控制指令会影响特定标志位。
- 可以通过转移指令来判断标志位的变化，实现程序中的分支结构或者循环结构。
- **标志寄存器**是实现程序中**分支、循环结构必须的硬件基础**。

状态标志位

- **以下的标志位属于状态标志位，**是指令根据执行情况为后续指令留下的一些可供参考的状态信息。（CF, OF, PF, AF, SF, ZF）



- 功能上，它们是实现分支、循环程序结构的基础。

- 器件上，它们的改变依赖于运算器（ALU）。

状态标志位有效的条件

- **状态标志位有效**的前提条件：
- 1）当前指令要影响标志位
- 2）当前指令对标志位的影响是有意义的
- 3）程序设计者对操作数的解释符合指令对该标志位的有效定义

进位标志CF

- **有效性**：在CPU进行算术运算指令时，如果**该指令**要影响CF标志，并且**用户**把操作数看作**无符号数（不完整编码除外）**，那么该标志位是**有效**标志。

- **含义**：它标志着上次算术运算最高位（字的第15位、字节的第7位）是否产生进位（加法指令）或者借位（减法指令）。

- 如果有进位或借位产生，那么CF=1；如果没有，那么CF=0。CF标志位位于FR的第0位。

进位标志CF

- 例1．观察下面的加减法运算，判断CF标志取值。（操作数理解为无符号数，CF标志才有意义）

- 10110011
- \+ 01010001
- 1 00000100

- 加法运算后，最高位向更高一位产生了进位，CF应等于1。

进位标志CF

- 00110000
- \+ 00001101
- 0 00111101

- 加法运算后，最高位没有向更高一位产生进位，CF应该等于0。

进位标志CF

- 01010101
- \- 00111110
- 0 00010111

- 减法运算后，最后位没有向更高一位产生借位，CF应该等于0。

判断CF标志的经典错误方法

- 但是容易出现一种人为的对CF标志的错误判别方法，那就是用补码加法来实现减法运算。

- 对上面的算式，这里尝试使用补码加法来实现它。

判断CF标志的经典**错误**方法

- 减数补码：　　00111110
- 相反数补码：　　11000010

- 如果使用这种方法来判断CF标志的取值会出现什么样的结果？

判断CF标志的经典**错误**方法

- 01010101
- \+ 11000010
- 1 00010111

- 运算结果和上面的减法算式完全一致。

- 但是最高位向更高一位产生了进位，按照CF的定义，CF却应该等于1，和减法算式**判别的结果正好相反**。

判断CF标志的经典**错误**方法

- 为什么会出现这样的情况，哪一种方法是正确的？

- 因为在这里使用了补码逻辑来判断无符号数的进位和借位，而**补码本身是针对有符号数产生的一种编码**。

- 使用补码来判断无符号数的运算过程本身就是不符合逻辑的。

判断CF标志的经典**错误**方法

- 在计算机硬件逻辑中，减法确实是使用补码加法来实现的，但是不能把这种实现方法和CF标志位的生成逻辑（生成CF标志值的逻辑电路结构）混为一谈。

- 在**手工判断CF标志位**的取值时，必须**按照常规的二进制数加减法运算**过程来进行，不能使用补码。

CF标志位的实用价值

- a. 如果需要进行**多字节无符号数或补码**的算术运算，那么CF标志就是低位字节和高位字节间进位和借位的桥梁。

- 8086/8088CPU提供的指令，能够直接处理的最长编码就是一个字（16位），如果超出这个范围，就必须使用多字节来表示要计算的数据。

- 任何CPU芯片，无论它处理的数据范围多么大，它总是一个有限的单位，如果超出这个单位，就必须使用标志位作为运算的中介。

CF标志位的实用价值

- b. 在执行移位指令时，CF标志用于存放移出位的值。
- 例如对01010011实行逻辑右移1位，即把这个字节中的每一位向右移动一位，左边空出的那一位置为0，最右边被移出的位保存在CF中。
- 这个例子中，移位完成后，CF应该等于1。

CF标志位的实用价值

- c. CF标志位能够为条件转移指令提供判别依据。

- 例如JC指令，它先判别CF标志位，如果CF=1，就跳转到指令中给出地址继续执行程序，如果CF=0，就不作跳转，CPU会顺序执行下一条指令。

- 在程序中，可以根据CF标志取值的不同来实现程序的分支或循环结构。

奇偶标志位PF

- 奇偶标志位PF（Parity Flag）：如果CPU所执行的指令要影响PF标志，并且该指令得到的数据结果低8位中含有偶数个“1”时，PF=1；含有奇数个“1”，PF=0。

- 注意无论指令的操作数有多么长，只有低8位数据中1的个数能够影响到PF标志的取值。

- PF标志位位于FR的第2位。

PF标志位的实用价值

- 这里以对ASCII码的校验来说明它的应用。

- ASCII码占用一个字节，但是只有低7位是真正的码值，最高位（第7位）是校验位。

PF标志位的实用价值

- 如果使用奇校验，那么必须保证编码字节中始终保持有奇数个“1”。
- 编码格式可以通过调整第7位的取值来实现：
- 如果ASCII码中有奇数个“1”，第7位取值为0；
- 如果ASCII码中有偶数个“1”，第7位取值为1；

PF标志位的实用价值

- 试设想计算机通过网络接收ASCII码，如何判断收到的编码是否正确呢？
- 判断收到的字节中是否为奇数个“1”，如果为偶数个“1”，收到的编码一定是错误的，可要求重传。
- 通过PF标志可以使用条件转移指令来实现程序中的分支或循环结构。
- 通过奇偶校验码能够识别大部分经常出现的错码，但是并不能完全避免错码。

辅助进位标志位AF

- ３）辅助进位标志位AF（Auxiliary Carry Flag）：在CPU执行算术运算指令时，如果该**指令**要影响AF标志，并且**用户**把操作数看作无符号数，AF标志才有意义。

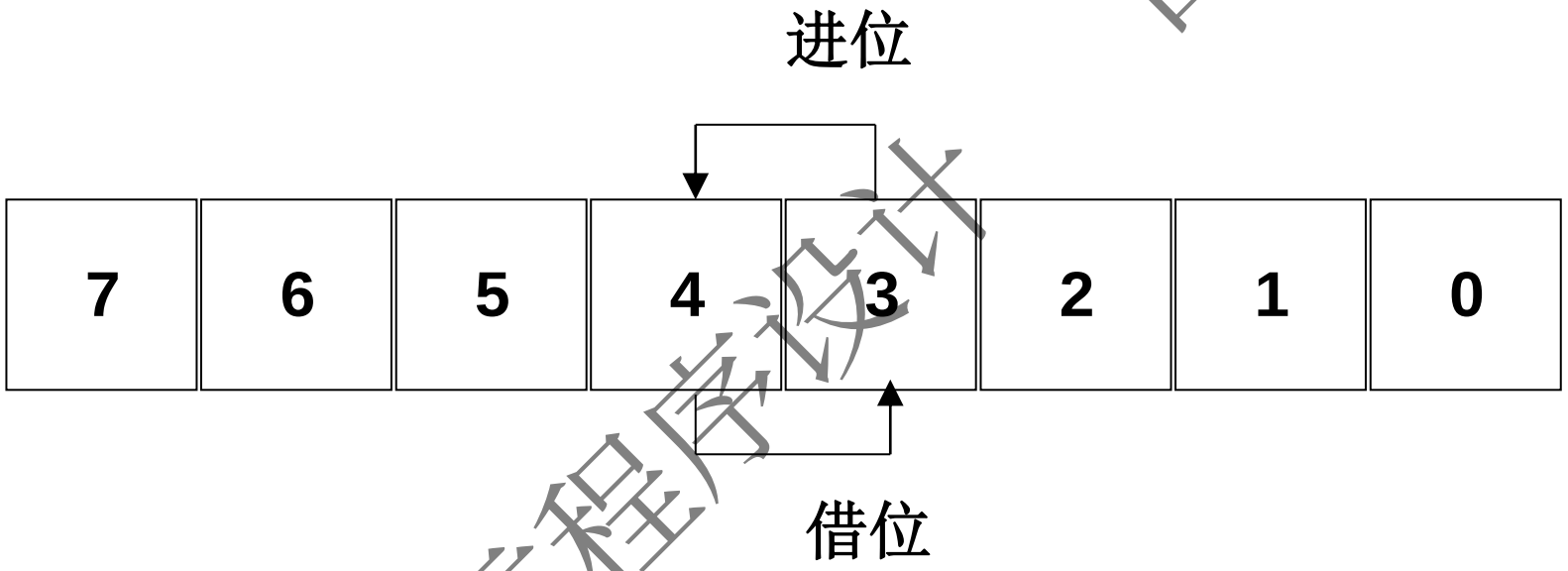
- 如果低字节中的低4位向高4位产生进位或借位时（第3位向第4位产生进位或借位），AF被置为1；否则AF被置为0。

辅助进位标志位AF

- AF标志，又称半进位标志，位于FR的第4位。

- 判别标准：和CF一样，使用无符号数的加减运算来作判断，只是判断进位和借位的位置不在操作数的最高位，而是在低字节的第3位。

辅助进位标志位AF



AF标志的实用价值

- AF标志的使用主要针对用二进制算术运算实现十进制算术运算的功能，单纯的二进制运算中，几乎不使用AF标志。

- BCD码：二进制数表示十进制数的编码，把4个连续二进制数位看作1个十进制数位，但实际上是一个十六进制数位，在运算中使用十六进制进借位规则。

AF标志的实用价值

- 如果能通过某种调整机制把进借位规则变为十进制进借位规则，则可使用二进制运算来实现十进制运算。

- AF标志位对于实现这种调整机制是必不可少的，因为它就是用于表达一个字节中低4位向高4位的进位或借位情况的。

零值标志位ZF

- 4）零值标志位ZF（Zero Flag）：如果CPU执行的**指令**要影响ZF标志，并且**保证ZF标志是有意义的**，ZF才具有意义。

- 当指令得到的结果数据各位全为“0”时，则ZF置“1”，否则ZF置“0”。

- ZF标志位位于FR的第6位。

ZF标志的实用价值

- 实用价值：ZF标志的使用主要是进行比较，并根据比较的结果来进行程序的分支或循环。例如比较两个整数是否相等。
- 比较的范围可以很广阔，不仅限于无符号数、补码，还可以对字符或者某些特定的编码进行比较。
- 和ZF标志相关的条件转移指令也很多，具体的应用示例在以后的课程中会遇到很多。

符号标志位SF

- 5）符号标志位SF（Sign Flag）：如果CPU执行的**指令**要影响SF标志，并且**用户**把操作数看作**带符号数（完整编码，或包括编码最高位）**，该标志位才有意义。
- 当计算结果为负数时，SF置“1”；当结果为正数时，SF置“0”。
- SF标志位的取值和结果数据的最高位是一致的，因为补码的最高位就是符号位。（**SF总是正确吗？**）
- SF标志位位于FR的第7位。

SF标志的实用价值

- 主要用于对算术运算结果的符号进行判断，只有针对带符号的运算才是有意义的。

- 可以通过两个带符号数经过减法运算后所得结果符号来判断哪一个数更大或者更小（需和OF结合使用），从而结合条件转移指令来实现程序的分支或循环。

溢出标志位OF

- （6）溢出标志位OF（Overflow Flag）：如果CPU执行的指令要影响OF标志，并且**用户**把操作数看作**带符号数（完整编码，或包括编码最高位）**时，OF标志位的取值才有意义。

- 如果运算结果超出了补码的表示范围（字节：-128~127，字：-32768到32767），那么解释为溢出，OF置为1；否则，OF置为0。

- OF标志位位于FR的第11位。

OF标志位的取值逻辑

- a. 正数+负数： OF=0 （等价运算：正数-正数； 负数-负数）

- 这种情况一定没有溢出，负数和正数是相互抵消的，运算结果比两个原始数据中绝对值较大的一个更靠近数轴的原点，所以一定不会超出定义的补码表示范围。

OF标志位的取值逻辑

- b. 正数+正数：
- 若结果为正数，表示符号位未丢失，无溢出：OF=0；
- 若结果为负数，表示符号位已经丢失，溢出：OF=1；
- （等价运算：正数-负数）

OF标志位的取值逻辑

- c. 负数+负数：
- 若结果为负数，表示符号位未丢失，无溢出：OF=0；
- 若结果为正数，表示符号位已经丢失，溢出：OF=1；
- （等价运算：负数-正数）

- 带符号数的加减运算可能出现的各种组合都可以等价替换为上面三种情况。

OF标志位的取值逻辑

- 因为出现溢出时：
- 1）参加运算的两个原始补码都没有超过表示范围
- 2）如果运算中超出了表示范围，只会是符号位**超出了表示范围一位**（**只能是一位，不会多于一位**）

OF标志位的取值逻辑

- 对8位操作数，两个最大的正数补码相加：

- 01111111
- + 01111111
- 0 11111110

- 可见符号位丢失，但只需在运算结果最高位前增添一个符号位，即为正确结果

OF标志位的取值逻辑

- 对8位操作数，两个最小的负数补码相加：

- 10000000
- + 10000000
- 1 00000000

- 若将进位作为最高位，则为正确结果

OF标志位的取值逻辑

- 3）所以符号位丢失与否的问题和溢出问题是等价的

- 4）硬件逻辑就是通过判断符号位丢失与否来判断补码运算是否溢出。

OF标志的实用价值

- 主要用于实现程序分支或循环；
- 8086/8088指令系统中有单独使用OF标志位的条件转移指令，可以实现对溢出现象的判断和分支处理；
- 也有结合使用OF标志和SF标志的条件转移指令，可以根据两个带符号数比较大小的结果来实现分支处理。

控制标志位

- 控制标志位不同于状态标志位。
- 功能上，它不用于实现分支、循环结构，而是设置CPU的工作模式或对特定事件的响应方式。
- 器件上，它们不依赖于ALU。

单步（或跟踪）标志位TF

- 7）单步（或跟踪）标志位TF（Trace Flag）：
- TF标志位是一个**控制标志位**，用于**触发单步中断**。

单步（或跟踪）标志位TF

- 如果使用指令将TF标志位置为1，那么CPU将进入单步执行指令的工作方式。
- 每执行完一条指令就会触发单步中断，执行单步中断服务程序。
- 一般在屏幕上显示CPU内部各寄存器和标志位状态，以便观察该指令产生的影响。

TF标志的实用价值

- 进入中断时，TF标志自动被清0，中断服务程序执行不会单步执行，中断服务程序结束后TF标志恢复中断以前的设置。
- 如果使用指令将TF标志清0，那么将会使CPU退出单步运行模式，回到连续执行机器指令的状态。TF标志位位于FR的第8位。
- 实用价值：TF标志为在机器指令级别上单步调试程序提供了相应的硬件基础。

中断标志位IF

- （8）中断标志位IF（Interrupt-enable Flag）：用于控制CPU是否处理可屏蔽中断。
- 如果使用指令将IF标志置为1，那么CPU将会处理任何可屏蔽中断；
- 如果使用指令将IF标志置为0，那么CPU将不会处理可屏蔽中断。
- IF标志位位于FR的第9位。

IF标志的实用价值

- 作为一个可设置开关，控制CPU是否处理可屏蔽中断，在程序执行不允许被打断的情况下，可以采用这个开关来屏蔽可屏蔽中断。
- 例如，某些对执行时间要求非常严格的程序段，执行这种程序段时不允许被中断打断，因为中断处理会添加额外的处理时间。
- 注意，IF标志只能屏蔽可屏蔽中断，对于一些由严重错误或故障引起的不可屏蔽中断则是无法屏蔽的。

方向标志位DF

- （9）方向标志位DF（Direction Flag）：这也是一个控制标志位，用于控制串操作指令存取数据的方向.

- 如果使用指令将DF标志置为0，每执行完一次串操作以后，源串地址指针SI和目的串地址指针DI中的内容会自动递增；

- 如果使用指令将DF标志置为1，那么每执行完一次串操作以后，SI和DI中的内容会自动递减。

DF标志的实用价值

- 使用该控制标志，可以由用户来选择串操作指令存取数据的方向。

- 串操作指令是一种特殊的指令，它能对一组相同类型的数据作相同的处理，比使用循环结构的程序效率更高，因为串操作指令的循环是在指令内部完成的，而循环结构的程序要完成循环需要执行多条指令。

标志位小结

- 条件转移指令可以对标志寄存器FR中的状态标志位加以判断，从而通过这些标志位**实现程序的分支或循环结构**。

- 关于哪些指令具体会对哪些标志位产生何种影响，以及如何在程序设计中正确的使用标志位，将会在后面的课程中逐步熟悉起来。

寄存器分类

- 地址指针寄存器（IP除外）和数据寄存器统称为**通用寄存器**，总共有8个，除了各自特殊的功能外，它们都可以用于存放数据；

- 指令指针和标志寄存器统称为**控制寄存器**，指令指针直接控制程序的执行流程，标志寄存器可以由标志位间接影响程序的执行流程；

- 段寄存器用于指示当前运行程序中可以使用的4个当前段。

寄存器的独特性

- CPU内部的寄存器数量很少，为了完成种类繁多的指令功能，CPU的设计者为每个寄存器都指定了它们独特的功能。
- 各寄存器的使用细节各不相同。这样的设计方法也是为了降低CPU硬件逻辑的复杂程度。
- 例如寄存器AL、AX分别可以用于乘法指令中存放乘数和乘积，其它寄存器就不能完成这一特定功能，这是8086/8088CPU硬件逻辑事先规定好的。

通用寄存器的隐含和特定使用

- 只要对这些寄存器的特殊功能加以记忆，再加上汇编程序的语法规则，这些寄存器的用法是可以熟悉的。
- 特别注意通用寄存器的**隐含**和**特定**使用。
- 当指令对一个寄存器**隐含使用**时，在指令中看不到该寄存器名称（地址），但指令有可能修改寄存器的内容，如果程序员忽略了这一点，则可能造成逻辑错误。

寄存器的隐含使用

- PUSH　AX　　；隐含使用SP
- POPF　　；隐含使用FR
- MUL　BL　　；隐含使用AL，AX
- LOOP　L1　　；隐含使用CX，IP
- AAA　　；隐含使用AL，AH

寄存器的特定使用

- SHL　AL，CL　　；特定使用CL
- IN　　AL，54H　　；特定使用AL
- OUT　DX，AL　　；特定使用DX，AL

- 如果忽略寄存器的特定使用，则造成语法错误。

第二章　微处理器的基础知识

存储器管理

存储器

- 存储单位

*1、二进制位：*信息的基本存储单位，可用小写字母b表示。

*2、字节：*信息的基本存取单位，可用大写字母B表示。一个字节由八位二进制数组成，占用一个存储单元。其位编号自左至右为$b_7b_6b_5b_4b_3b_2b_1b_0$。

- *3、字：*一个字16位，占用两个存储单元。其位编号为b_{15}～b_0。

- *4、双字：*一个双字32位，占用四个存储单元。其位编号为b_{31}～b_0。

- *5、四字：*一个四字64位，占用八个存储单元。其位编号为b_{63}～b_0。

- **存储顺序**：逆序存放

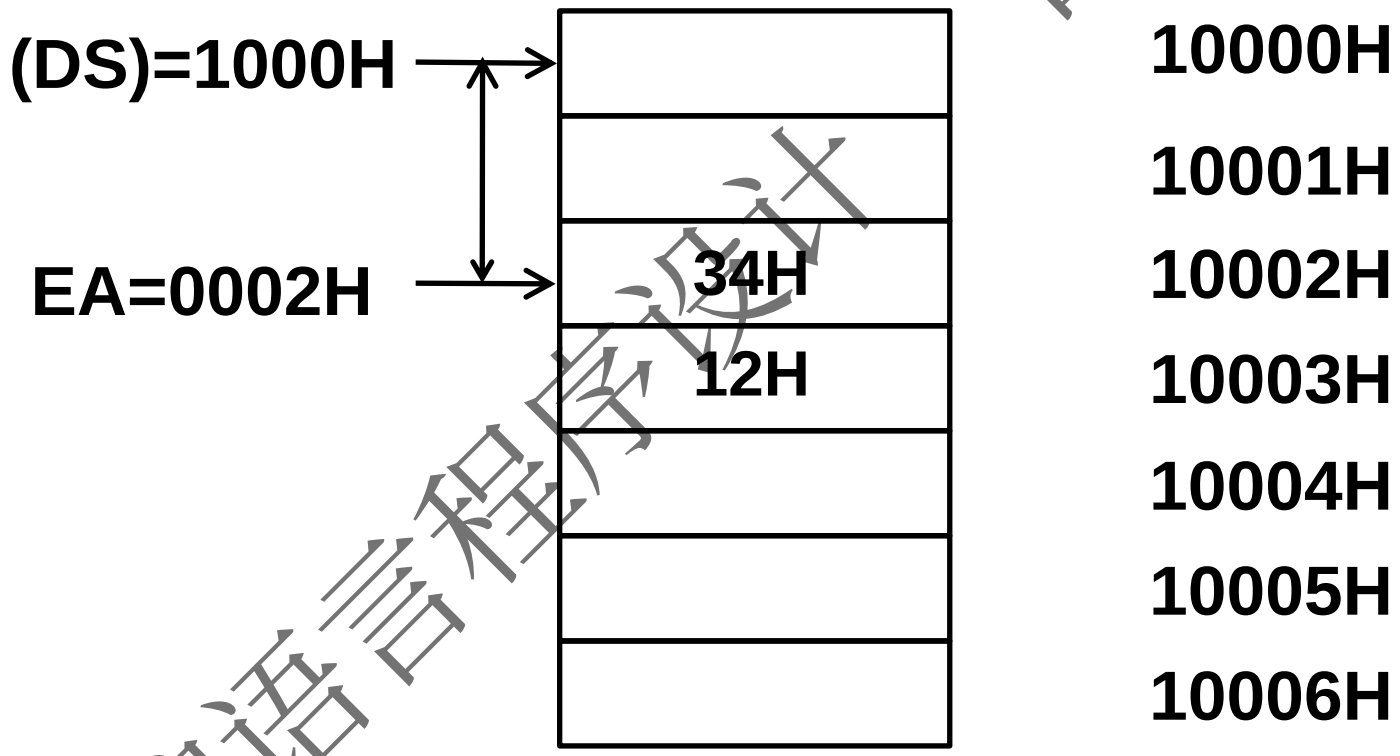
- 照Intel公司的习惯，对于字、双字、四字数据类型，其低地址中存放低位字节数据，高地址中存放高位字节数据。

- 这就是 “逆序存放”的含义。

存储顺序

- 示例：
- MOV DS:[0002H], 1234H
- ; (DS) = 1000H

存储顺序



- **存储地址组织**：存储器分段管理

- IBM PC机的存储器采用分段管理的方法。

- 一个内存单元地址使用段基值和偏移量两个逻辑地址分量来描述，表示为段基值:偏移量。

- *对段基址的限定：*实模式下，段基址必须定位在地址为16的整数倍上，即20位地址的低4位为0。

- 最多可以定义多少个段？

- 一个段最长为多少字节？

- 存储器采用分段管理后，其物理地址的计算方法为：

- 10H×段基值＋偏移量
- （其中H表示是十六进制数）

- *例.* 某内存单元的地址用十六进制数表示为1234H:5678H，则其物理地址为12340H＋5678H＝179B8H。
- 　　如图2-5所示。

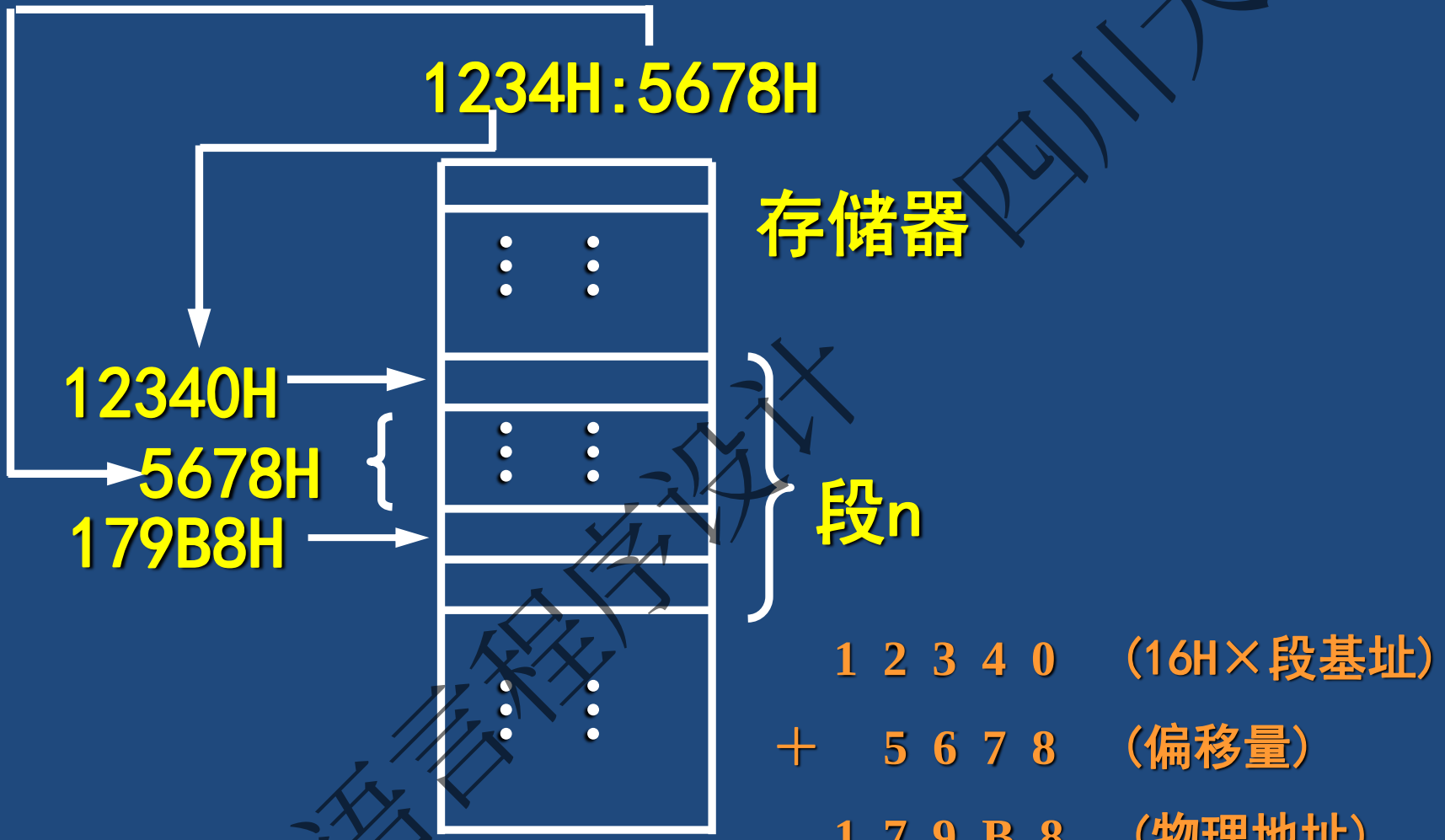


图2-4 物理地址的形成

堆栈

- 堆栈段是以段寄存器**SS指明段基址**，由寄存器**SP指明栈顶数据偏移量**，以**字为基本存储单位**，存取操作遵循**后进先出**原则的一种段。

- 针对堆栈段的两种操作：把字数据送到栈顶保存，称为**入栈**；把栈顶字数据保存到指定位置，这称为**出栈**。

- 注意，入栈和出栈操作都必须是在栈顶完成的。

堆栈的重要功能

- （1）调用子程序时返回点的保存和恢复
- （2）中断调用时断点的保存和恢复
- （3）执行现场的保存和恢复

（1）调用子程序时返回点的保存和恢复

- 无论是高级语言还是汇编语言，在任何实用的程序中，都会使用子程序来完成一些相对独立的功能。

- 在汇编语言中调用子程序使用CALL指令，子程序返回主程序使用RET指令。

（1）调用子程序时返回点的保存和恢复

- 例1．堆栈对于子程序调用的作用
- 主程序片段：
- …
- …
- CALL PROC1
- #主程序中使用CALL指令调用子程序PROC1，即跳转到PROC1的起始地址
- …
- …

（1）调用子程序时返回点的保存和恢复

- 子程序PROC1片段：
- …
- …
- RET
- #子程序执行完毕后使用RET指令返回主程序

（1）调用子程序时返回点的保存和恢复

- 按照程序流程，CALL指令调用子程序后，必须返回到CALL指令后面那条指令的位置继续执行。

- 那么在执行CALL指令时CPU必须记载紧跟它后面那一条指令的地址，否则以后就无法返回主程序。

（1）调用子程序时返回点的保存和恢复

- CALL指令在实行跳转前，会自动把当前的CS、IP寄存器中的内容按先后顺序入栈保存。

- 此时CS、IP中的逻辑地址是指向CALL指令后面那条指令的（在取指周期中，IP已经被修改，指向了下一条指令）。

- 这就是返回点的保存。

（1）调用子程序时返回点的保存和恢复

- CALL指令执行跳转后，程序流程跳转到子程序PROC1的起始地址，CPU开始执行子程序，子程序执行完后，会使用RET指令返回先前的返回点。

- 执行RET指令时，会从栈顶出栈两个字数据按顺序恢复到IP、CS寄存器中，这就是返回点的恢复。

（1）调用子程序时返回点的保存和恢复

- 返回点恢复后程序流程回到主程序CALL指令后面那条指令继续往下执行。

- 当然，在执行RET指令时，必须保证堆栈的栈顶数据和执行CALL指令后的情况一致。

- 如果在执行RET指令前又向栈顶压入了新数据并且未出栈，那么RET指令将会使程序流程转移到错误的地址。

（1）调用子程序时返回点的保存和恢复

- 堆栈是实现子程序调用和返回机制的硬件基础。

- 如果没有堆栈这种硬件机制的支持，子程序调用与返回都是无法实现的。

（2）中断调用时断点的保存和恢复

- CPU在执行程序时，经常会有来自设备接口或者CPU内部的中断信号，请求CPU执行相应的中断服务程序处理中断请求。

- 如果CPU处理一个中断请求，就会暂停当前程序的执行，在当前指令执行完毕后就调用相应的中断服务程序。

- 中断服务程序执行完毕后，会使用IRET指令返回到先前执行程序被打断的位置继续执行。

（2）中断调用时断点的保存和恢复

- 在执行中断调用以前，CPU会把当前CS、IP、FR中的内容入栈，这个过程就称为断点保存，为以后恢复当前程序的执行作准备。

- 中断服务程序执行完毕后，会使用IRET指令从堆栈中出栈3个字数据，按照与入栈时相反的顺序分别恢复到FR、IP、CS中，这个过程就称为断点恢复。

（2）中断调用时断点的保存和恢复

- 与子程序调用比较，中断调用多保存了标志寄存器FR的内容。
- 子程序调用的时刻是程序员决定的，如果程序员认为有必要保存FR，那么可以自己在程序中实现。
- 但是**中断调用的时刻不是程序员能够预知的**，它可以发生在任何时刻，CPU执行程序中任何一条指令时，都有可能发生中断。

（2）中断调用时断点的保存和恢复

- 前一条指令留下的标志位信息可能会直接影响到后续指令的执行结果，不能因为执行中断服务程序破坏FR的本来面目。

- 因此，中断调用比一般的子程序调用多保存了FR中的内容。

（2）中断调用时断点的保存和恢复

- 堆栈也是中断调用与返回机制的硬件基础。

- 如果没有堆栈的支持，中断调用和返回同样是无法实现的。

（3）执行现场的保存和恢复

- CPU执行现场：CPU当前的所有寄存器、标志位的状态。

- 如果子程序在主程序未知的情况下任意破坏执行现场，则会在主程序中导致逻辑错误。

- 例2．子程序中，CPU执行现场的保护与恢复。

（3）执行现场的保存和恢复

- 子程序：
- PUSH AX
- PUSH BX
- PUSH CX
- PUSH DX
- …
- POP DX
- POP CX
- POP BX
- POP AX
- RET

（3）用户临时保存和恢复数据

- 这个子程序中要使用AX、BX、CX、DX，但是程序员**不希望**在返回主程序以后这四个寄存器中的内容**发生改变**。

- 程序中入栈和出栈的顺序是相反的，最先入栈的寄存器内容最后出栈，这种操作方式是堆栈后进先出的规则决定的。

- 使用变量来保存临时数据，不如堆栈方便。

2.6 外部设备及I/O地址空间

- 8086/8088所提供的I/O地址总是16位的，所以允许最大的I/O寻址空间为64KB，寻址范围为0000H～FFFFH。

- 8086/8088中，访问内存单元与I/O端口使用不同的控制信号，在指令系统中体现为不同的指令。