3．1 8086/8088指令的构成

- 指令寻址方式
- 机器指令构成与汇编指令格式

3．1．1 指令的寻址方式

- 指令的**寻址方式**是指**指令获取操作数的方式。**

- 寻址方式总是针对具体某个操作数而言，如果一条指令中各个操作数的寻址方式不同，那么这条指令就同时具有多种寻址方式。

（1）寄存器寻址（Register Addressing）

- 如果一条指令中的某个操作数地址是寄存器，那么称该指令按照寄存器寻址获取该操作数。

（1）寄存器寻址（Register Addressing）

- 例1. 指令中使用寄存器寻址方式获取操作数
- MOV AX，BX
- ＡＸ＜＝（ＢＸ）
- BX为源操作数地址，AX为目的操作数地址，两个操作数地址均为寄存器，寻址方式均为寄存器寻址。

（1）寄存器寻址（Register Addressing）

- 寄存器寻址的特点：寄存器在CPU内部，不需要执行总线周期，操作数读写速度很快。

（2）立即数寻址（Immediate Addressing）

- 如果指令所需要的操作数直接在指令代码中，称指令按照立即数寻址方式获取该操作数。

- 立即数寻址方式只能针对源操作数，不能针对目的操作数，因为**立即数是数据本身，不是存储单元地址**。

（2）立即数寻址（Immediate Addressing）

- 例2.　在指令中使用立即数寻址方式获取源操作数

- MOV CX，0B3CAH

- ＣＸ＜＝０Ｂ３ＣＡＨ

- 0B3CAH是源操作数，寻址方式为立即数寻址，CX为目的操作数地址，寻址方式为寄存器寻址。

（2）立即数寻址（Immediate Addressing）

- MOV BX，“AB”
- 把A、B两个字符的ASCII码保存到寄存器BX中，
- “AB”字符串为源操作数，寻址方式为立即数寻址，
- BX为目的操作数地址，寻址方式为寄存器寻址。

（2）立即数寻址（Immediate Addressing）

- 立即数寻址的特点：在CPU的取指周期就把操作数随同指令一起读取到指令队列，
- 指令执行时获取操作数只需要直接从指令队列中读取。
- 不需要再另外启用总线周期从内存单元读取，获取操作数的速度很快。

（3）存储器寻址

- 寄存器寻址针对的操作数在寄存器中，不是针对内存操作数。

- 立即数寻址针对的操作数在指令中，不能把它和一般的内存操作数混淆起来。

（3）存储器寻址

- 针对内存单元的各种寻址方式统称为存储器寻址。
- CPU读写内存操作数必须在取指周期之后重新启用总线周期。
- 使用存储器寻址方式读写操作数比前面介绍的两种寻址方式慢。

（3）存储器寻址

- 内存单元的**逻辑地址**由**段基值**和**偏移量**两部分构成。

- 段基值由特定段寄存器给定，在汇编指令中使用内存操作数一般只需给出偏移量，**与之搭配的段寄存器由汇编程序默认确定**。

- 程序员也可以在汇编指令中显式给出所用段寄存器。

（3）存储器寻址

- 讨论存储器寻址，就是**讨论偏移量的各种构成方法**。总的来说偏移量可以取三种分量之和：
- （a）位移量：指令中的位移量字段，是一个8位或16位的二进制编码。
- （b）基地址（基址）：基址寄存器BX或基址指针BP中的内容。
- （c）变地址（变址）：变址寄存器SI或DI中的内容。

（3）存储器寻址

- 通过这三个分量的不同组合并相加，可以得到不同的EA获取方式，即不同的存储器寻址方式。

- 需要注意的是，在通常情况下，EA分量都理解为无符号数。

- 转移指令具有带符号位移量，带符号位移量是针对代码段中指令的访问，不是针对指令中操作数的寻址。

1）直接寻址方式（Direct Addressing）

- 指令使用位移量作为内存单元的偏移量，这种寻址方式称为直接寻址方式。

- 在汇编语言源程序中，直接寻址方式的偏移量可以用数值（**数值位移量**）或符号（**符号位移量**）来表示。

- EA = Disp

1）直接寻址方式（Direct Addressing）

- 例1．在指令中对操作数使用直接寻址方式
- MOV BX, DS:[1000H]
- MOV BX, [1000H]
- 两条指令功能一致，把从数据段段基址起偏移1000H个字节那个字单元中的内容传送到寄存器BX中

1）直接寻址方式（Direct Addressing）

- 第一种写法显式的指定段寄存器DS，
- 第二种写法隐含的使用段寄存器DS。
- 1000H是用数值表示的位移量，源操作数寻址方式为直接寻址方式，**位移量直接作为偏移量**。

1）直接寻址方式（Direct Addressing）

- MOV BX， VAR

- VAR是程序中定义的一个变量，变量对应于内存中的字节单元或字单元，引用变量名称即是引用内存单元的地址。

- **变量名称**实质上是**符号位移量**，在汇编过程中，它会被替换为数值位移量。

1）直接寻址方式（Direct Addressing）

- MOV DA_BYTE， 0FH

- 源操作数为立即数寻址，目的操作数为直接寻址，DA_BYTE为变量名称。

1）直接寻址方式（Direct Addressing）

- MOV CL， DA+3
- 目的操作数为寄存器寻址，
- 源操作数为直接寻址，DA是变量名称，+3是指由DA字节单元起向高地址端偏移3个字节后的字节单元。
- 在汇编过程中，DA+3会被换算为一个确定的数值位移量。

1）直接寻址方式（Direct Addressing）

- 无论使用数值地址还是符号地址，指令对于直接寻址方式隐含使用的段寄存器都是DS

- 如果所访问的内存单元不在数据段中，必须在指令中显式的指出段寄存器名称。

2）寄存器间接寻址方式（Register Indirect Addressing）

- 如果内存操作数的有效地址（偏移量，EA）直接由地址指针寄存器SI、DI、BX、BP之一中获得，操作数的寻址方式称为寄存器间接寻址。

- EA = （BX）
- EA = （BP）
- EA = （SI）
- EA = （DI）

2）寄存器间接寻址方式（Register Indirect Addressing）

- 例1．在指令中对操作数使用寄存器间接寻址方式
- MOV CH，[SI]
- 将(SI)指示的字节单元中的数据传送到寄存器CH
- 源操作数为寄存器间接寻址，隐含使用的段寄存器为DS

2）寄存器间接寻址方式（Register Indirect Addressing）

- MOV [BP]，CX

- 目的操作数为寄存器间接寻址，有效地址由(BP)指出，隐含使用的段寄存器为SS

- 要特别注意隐含搭配的段寄存器。

3）变址寻址（Indexed Addressing）和基址寻址（Based Addressing）

- 两种寻址方式都针对内存操作数，有效地址由地址指针寄存器（BX，BP，SI，DI其中之一）中的内容和指令中给出的位移量两部分相加得到。

3）变址寻址（Indexed Addressing）和基址寻址（Based Addressing）

- 变址寻址：使用变址寄存器SI或DI中的内容和指令中的位移量相加得到EA，那么称为变址寻址。

- EA = （SI）+ Disp

- EA = （DI）+ Disp

3）变址寻址（Indexed Addressing）和基址寻址（Based Addressing）

- 基址寻址：使用基址寄存器BX或BP中的内容和指令中的位移量相加得到EA，那么称为基址寻址。

- EA = （BX）+ Disp

- EA = （BP）+ Disp

3）变址寻址（Indexed Addressing）和基址寻址（Based Addressing）

- 例1．在指令中使用变址或基址寻址
- MOV AX，10H[SI]
- 对源操作数使用变址寻址
- 把EA=（SI）+10H的字单元中的数据传送到AX寄存器中
- 10H为数值位移量，隐含使用的段寄存器为DS。

3）变址寻址（Indexed Addressing）和基址寻址（Based Addressing）

- MOV TAB1[BP]，CL

- 目的操作数为基址寻址，
- EA=（BP）+TAB1

- 隐含使用的段寄存器为SS，BP在寻址方式中默认与SS搭配使用。

4）基址变址寻址（Based Indexed Addressing）

- 内存操作数的有效地址为基址寄存器（BX或BP）的内容、变址寄存器（SI或DI）的内容与指令中的位移量（8位或16位）三部分分量之和时，称为基址变址寻址。

- EA = （BX）+（SI）+ Disp
- EA = （BX）+（DI）+ Disp
- EA = （BP）+（SI）+ Disp
- EA = （BP）+（DI）+ Disp

4）基址变址寻址（Based Indexed Addressing）

- 如果指令中没有显式给出段寄存器，那么隐含使用的段寄存器由所使用的基址寄存器来决定，BX默认搭配DS，BP默认搭配SS。

4）基址变址寻址（Based Indexed Addressing）

- 例1．指令中使用基址变址寻址
- MOV AX，200H[BX][SI]
- 源操作数为基址变址寻址

4）基址变址寻址（Based Indexed Addressing）

- 200H为数值位移量
- （BX）为基址分量
- （SI）为变址分量

- 三个分量之和为字单元的EA，把该字单元中的数据传送至AX寄存器保存。

- 隐含使用的段寄存器为DS。

4）基址变址寻址（Based Indexed Addressing）

- MOV TAB1[BP][DI]， DL

- 目的操作数为基址变址寻址，把寄存器DL中的数据传送至内存中的二维字节数组，

- TAB1指明表头偏移量，BP和DI中的内容理解为数组的浮动下标。

4）基址变址寻址（Based Indexed Addressing）

- 思考：

- 高级语言中的一维、二维数组的下标变量可以用寄存器来表示，那么多维数组的下标变量用何种存储单元表示？

4）基址变址寻址（Based Indexed Addressing）

- 容易误判的寻址方式：
- MOV　AL，[BX]
- MOV　AL，10H[BX]
- MOV　AL，[BX][SI]
- MOV　AL，20H[BX][SI]

（3）存储器寻址

- 存储器寻址通过取不同分量求和得到有效地址，形成了五种寻址方式。

- 特别注意默认段寄存器的使用，若内存操作数不在默认段内，那么必须在指令中明确的给出段前缀。

（3）存储器寻址

- MOV AX，ES：TAB[SI]

- 目的操作数为变址寻址，默认使用的段寄存器为DS，TAB变量却定义在附加段中。

- 所以必须在目的操作数前加上段前缀ES：，此时段前缀不能省略。

（4）串操作寻址方式

- 8086/8088指令系统提供了一组串操作指令，是无操作数指令，操作数为隐含操作数，源操作数和目的操作数都位于内存中。

（5）I/O端口寻址

- I/O端口寻址方式，是指指令如何计算端口的地址，地址计算方法的不同形成不同的寻址方式。

- 由于I/O空间和内存空间在8086/8088系统中被组织为两个独立的、无关的逻辑空间，所以访问I/O端口的指令是特定的，完全和访问内存单元的指令无关。

（5）I/O端口寻址

- 8086/8088的端口地址仅使用地址总线的低16位，无逻辑地址与物理地址的区分。

- 8086/8088的端口寻址范围是多大？

（5）I/O端口寻址

- I/O端口直接寻址方式：
- IN　AL，60H
- OUT　40H, AX
- 直接寻址方式只能使用8位端口地址，寻址范围为0~255。

（5）I/O端口寻址

- I/O端口间接寻址方式：

- MOV　DX, 0FF55H
- IN　　AL，DX

- 间接寻址方式可以使用16位端口地址，寻址范围为64 KB。

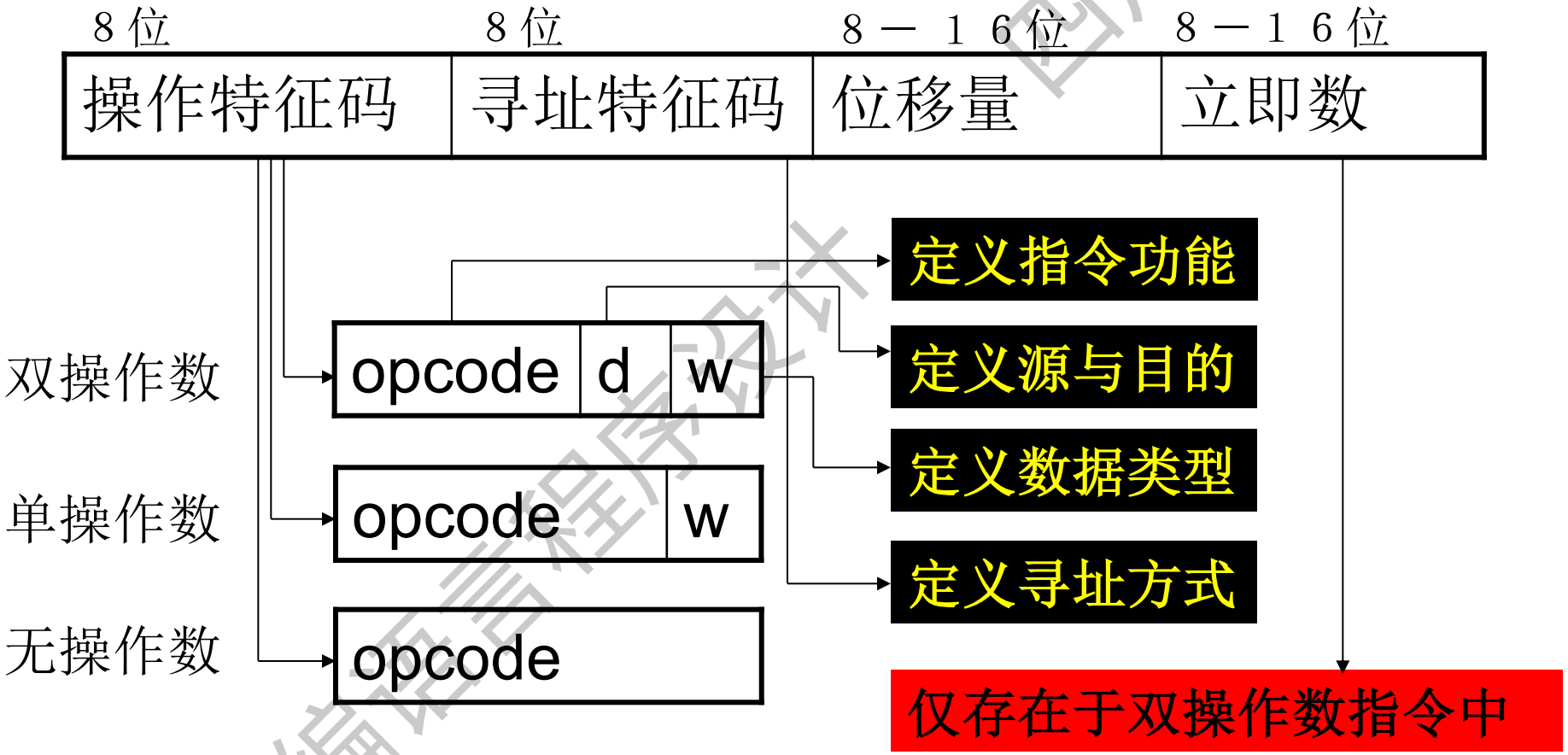
3．1．2 机器指令的构成

- 汇编指令和机器指令是一一对应的。
- 汇编指令的构成规则并不是随意确定的，是根据机器指令代码的构成规则演变而来的。
- 机器指令的构成仅要求粗略了解。

指令按操作数数量分类

- 按照指令中显式给出的操作数数量分类：
- 双操作数指令
- 单操作数指令
- 无操作数指令

机器指令的构成



（1）双操作数指令

- 格式：OPR DEST，SRC
- OPR　操作助记符，说明指令功能
- DEST　目的操作数地址
- SRC　源操作数或源操作数地址

（1）双操作数指令

- 8086/8088指令系统中的指令最多只能支持两个操作数，而且位于内存中的操作数最多只能有一个。

- 以上规则不包括隐含操作数。

- 错例：
- ＭＯＶ　［１０００Ｈ］，［ＢＸ］

（1）双操作数指令

- 立即数仅存在于双操作数指令中，且立即数仅能作为源操作数。
- 错例：
- ＡＤＤ　　１２，　　３４

（1）双操作数指令

- 例1．简要说明以下双操作数指令的格式

- MOV AX，BX

- 数据传送指令
- 功能： AX ＜＝（BX）

- MOV　　操作助记符
- AX　　目的操作数地址
- BX　　源操作数地址

（1）双操作数指令

- ADD AX，BX
- 加法指令
- 功能：AX ＜=（AX）＋（BX）

（1）双操作数指令

- ADD　　操作助记符
- AX　　源操作数地址，目的操作数地址
- BX　　是源操作数地址。

（1）双操作数指令

- SUB BX，157
- 减法指令
- 功能： BX ＜＝（ BX ）－１５７

（1）双操作数指令

- SUB　　操作助记符
- BX　　源操作数地址、目的操作数地址
- 157　　源操作数（不是操作数地址），
- 立即数是在指令代码中的数据，执行指令时CPU直接在指令队列中读取立即数。

（2）单操作数指令

- 单操作数汇编指令的格式：OPR DEST
- OPR是操作助记符，说明指令的功能
- DEST可能是源操作数地址，也可能是目的操作数地址。
- 单操作数指令中可能会使用隐含的操作数，机器指令的隐含操作数由OPCODE字段确定，并不由寻址特征来决定。

（2）单操作数指令

- 例2．简要说明下面单操作数指令的格式

- NEG AX

- 求相反数指令
- 功能： AX <= －（AX）

- NEG　　操作助记符

- AX　　源操作数地址，目的操作数地址。

（2）单操作数指令

- PUSH AX

- 压栈指令
- 功能：（SP）<=（SP）－2
  （SP）<=（AX）

- PUSH　操作助记符

- AX　源操作数地址

（2）单操作数指令

- 目的操作数是隐含的，是堆栈段当前栈顶的那个字单元。
- PUSH指令会使用堆栈中的存储单元作为目的操作数地址，由机器指令的OPCODE决定。
- 仅从指令的操作数特征观察，观察不到隐含操作数。

（2）单操作数指令

- 单操作数指令无立即数字段，在单操作数指令中不能使用立即数作为源操作数。

- 错例：
- PUSH 248

（3）无操作数指令

- 无操作数汇编指令的格式：OPR
- OPR是操作助记符，说明指令的功能
- 无操作数指令可能真的没有操作数，但也可能存在隐含操作数。
- 无操作数指令的代码总是固定为一个字节。

（3）无操作数指令

- 例3 简要说明下面无操作数指令的格式
- CLC
- 清除ＣＦ标志指令
- ＣＦ　＜＝　0

（3）无操作数指令

- CLC　　操作助记符

- 指令格式中没有任何操作数，隐含使用的操作数地址为FR中的CF标志位。

- 这种隐含操作数地址不由寻址特征码指出，而是由机器指令的OPCODE指出。

（3）无操作数指令

- NOP
- 空操作指令
- 功能：什么操作也不做，只是让CPU空转三个时钟周期（节拍）。
- 在汇编语言程序中，空操作指令可用于延时，也可以用在调试程序时清除其它指令的最好工具。

（3）无操作数指令

- POPF
- 功能：出栈数据送到标志寄存器FR
- 指令中没有显式的操作数，隐含操作数地址为标志寄存器和栈顶字单元。

（3）无操作数指令

- 无操作数指令仅含OPCODE字段，其它与操作数相关的字段都不需要了。

3．1．3 指令系统

- 8086/8088指令系统中的指令按照功能分类有六类：

- 传送类指令（Transfer Instruction）：功能为把数据从一个存储位置搬运到另一个存储位置，**一般不影响标志位，仅当FR作为目的操作数地址时会影响标志位**。

- 算术运算类指令（Arithmetic Instruction）：用于完成算术运算，**一般要影响标志位**。

3．1．3 指令系统

- 位操作类指令（Bit Manipulation Instruction）：主要用于逻辑运算、移位等功能，按二进制位对数据进行操作，**一般要影响标志位**。

- 串操作类指令（String Instruction）：主要用于对串数据进行循环操作。**影不影响标志位视情况而定**。

3．1．3 指令系统

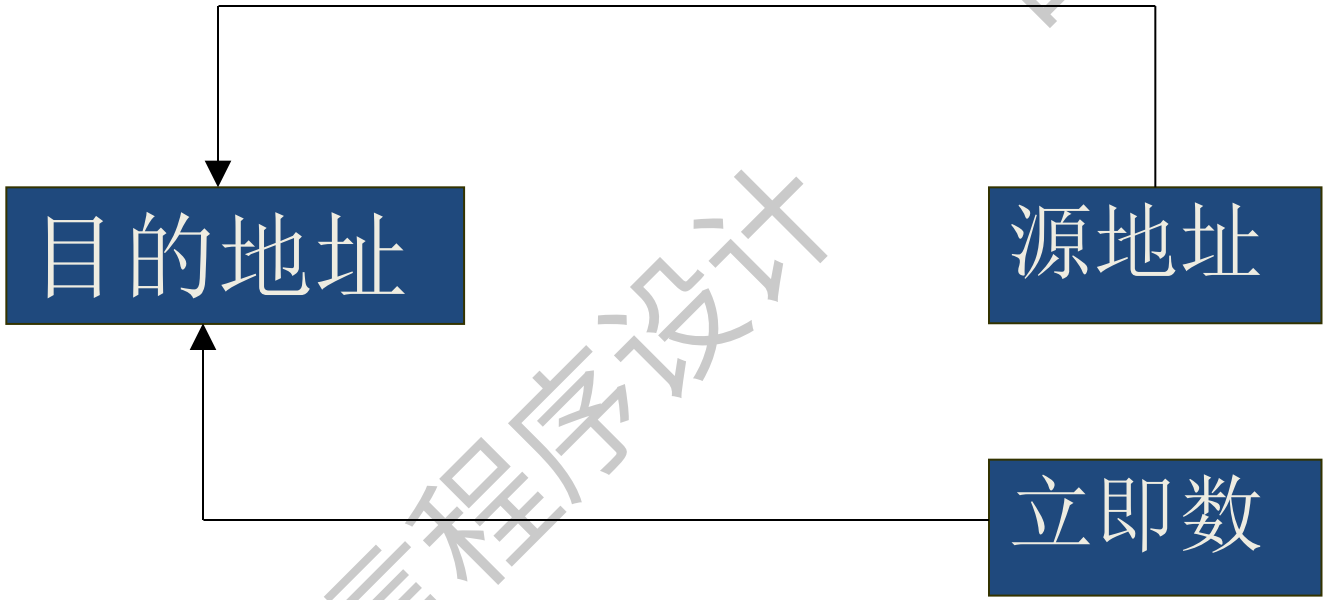
- 程序转移类指令（Program Transfer Instruction）：有条件的或无条件的修改指令指针IP（近跳转）或者IP和CS寄存器中的内容（远跳转），实现程序流程的中的分支、循环结构。**不影响标志位。**

- 处理器控制类指令（Processor Control Instruction）：主要是指用于设置标志位的指令、控制CPU运转的指令、空操作指令，其中的部分指令能够控制CPU的运转方式或CPU对某些特殊事件的处理方式。**影不影响标志位视情况而定。**

（1）传送类指令

- 1．传送类指令
- （1）数据传送指令（Move）
- 指令格式：MOV DEST，SRC
- 指令功能：DEST <= （SRC）
  DEST <= SRC
- 标志位影响：无

（1）传送类指令



（1）传送类指令

- 在数据传送指令中，可使用各种寻址方式，但两个操作数中，只能有一个为存储器寻址。

- 如果要实现两个内存单元间的数据传送，必须通过两条数据传送指令来实现，并且通过寄存器作数据传送中转。

（1）传送类指令

- 例1 把DA_WORD1字单元内容转送到DA_WORD2字单元中，可用如下两条指令：

- MOV AX，DA_WORD1
- MOV DA_WORD2，AX

（1）传送类指令

- MOV指令中立即数不能直接传送给段寄存器，而且段寄存器之间也不能直接传送数据，但可以通过通用寄存器来实现间接传送。

（1）传送类指令

- 例2 把立即数10A0H传送给段寄存器DS：

- MOV AX，10A0H
- MOV DS，AX

（1）传送类指令

- 错例：（下列指令中存在语法错误）

- MOV　VAR1, [1000H]
- MOV　[BX], 02H
- MOV　AL, 75A4H
- MOV　DS, 0241H
- MOV　ES, DS
- MOV　CS, AX

（2）交换指令（Exchange）

- 指令格式：XCHG DEST， SRC
- 指令功能：（DEST）⇔（SRC）
- 交换指令不能使用立即数，也不能使用段寄存器。
- 标志位影响：无

（2）交换指令（Exchange）

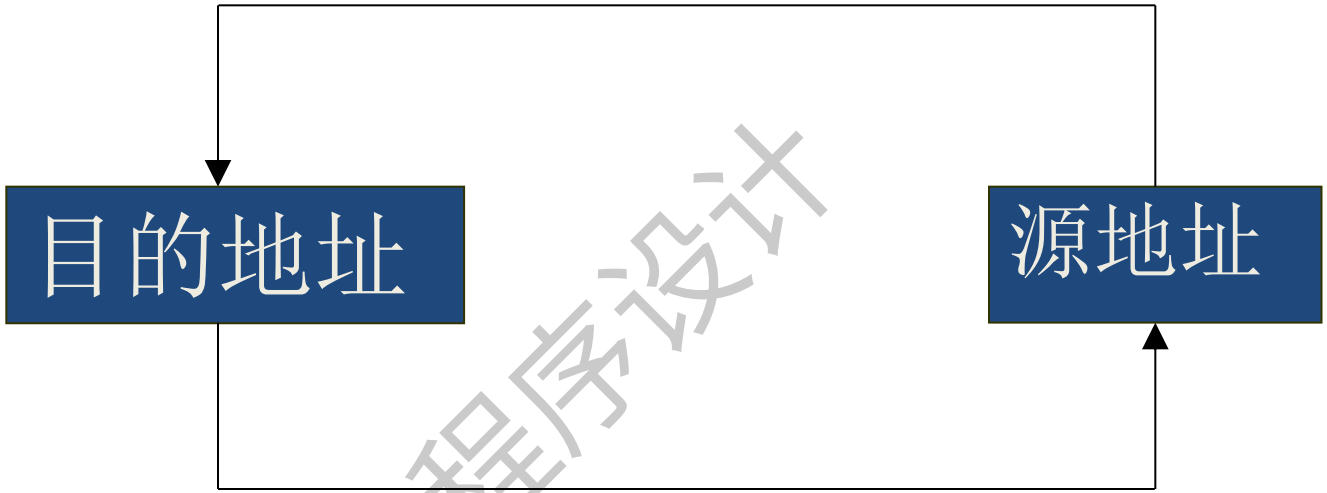
- 交换指令中也最多只能有一个内存操作数，两个内存单元中的内容不能直接交换。

（2）交换指令（Exchange）

- 例3 要将两个字节存储单元DAB1，DAB2的内容进行交换，可以用以下三条指令来实现：

- MOV AL，DAB1
- XCHG AL，DAB2
- MOV DAB1，AL

（2）交换指令（Exchange）



（3）堆栈操作指令

- 堆栈段的存取原则是“后进先出”，且以字（16位）为单位。
- 段基值存放于SS中；栈顶偏移量存放于SP中,SP始终指向栈顶。
- 堆栈主要用于对CPU现场的保护与恢复（ＰＵＳＨ与ＰＯＰ指令）、子程序与中断服务返回地址的保护与恢复等。

1）入栈指令

- 指令格式：PUSH SRC
- 单操作数指令，但是使用了一个隐含操作数，该隐含操作数在堆栈中。
- 指令功能：把源操作数压入堆栈
- 标志位影响：无

1）入栈指令

- PUSH指令的操作数可以是通用寄存器、段寄存器或者字存储单元中的内容。

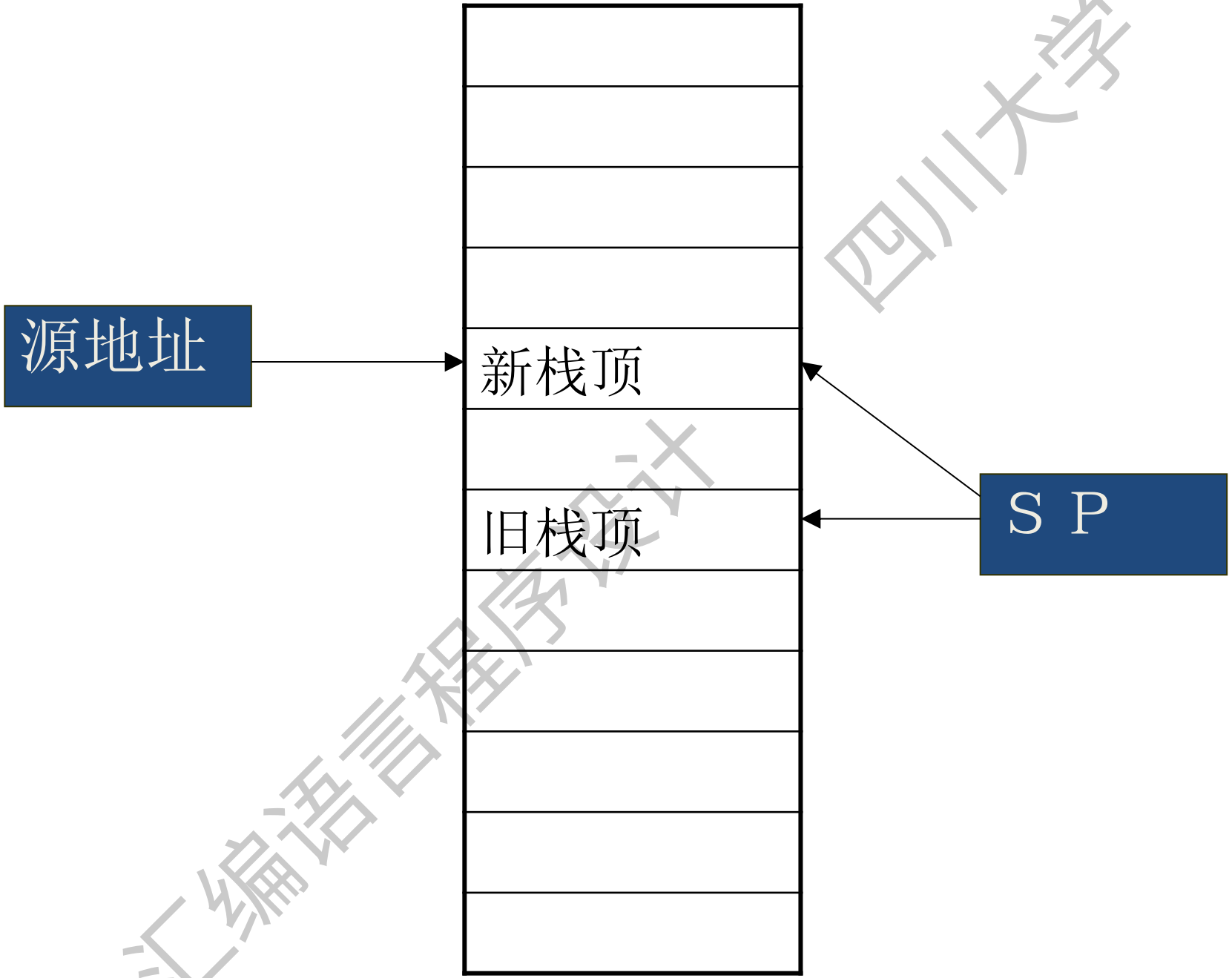
1）入栈指令

- PUSH指令的执行步骤如下：

- （SP）-2 –> SP
- 先修改栈顶指针，使它向低地址移动一个字单元（空的）

- (SRC) -> (SP)
- 把数据保存至SP指向的字存储单元

- 对于标志寄存器的入栈操作可以使用指令PUSHF，它没有显式操作数，FR中的内容作为隐含操作数。



2）出栈指令

- 指令格式：POP DEST
- 单操作数指令，使用了一个隐含的，位于堆栈中的操作数。
- 指令功能：把栈顶数据取出，保存到目的地址中。
- 标志位影响：无

2）出栈指令

- 操作数地址和PUSH指令一致，可以是通用寄存器、段寄存器或者字存储单元。

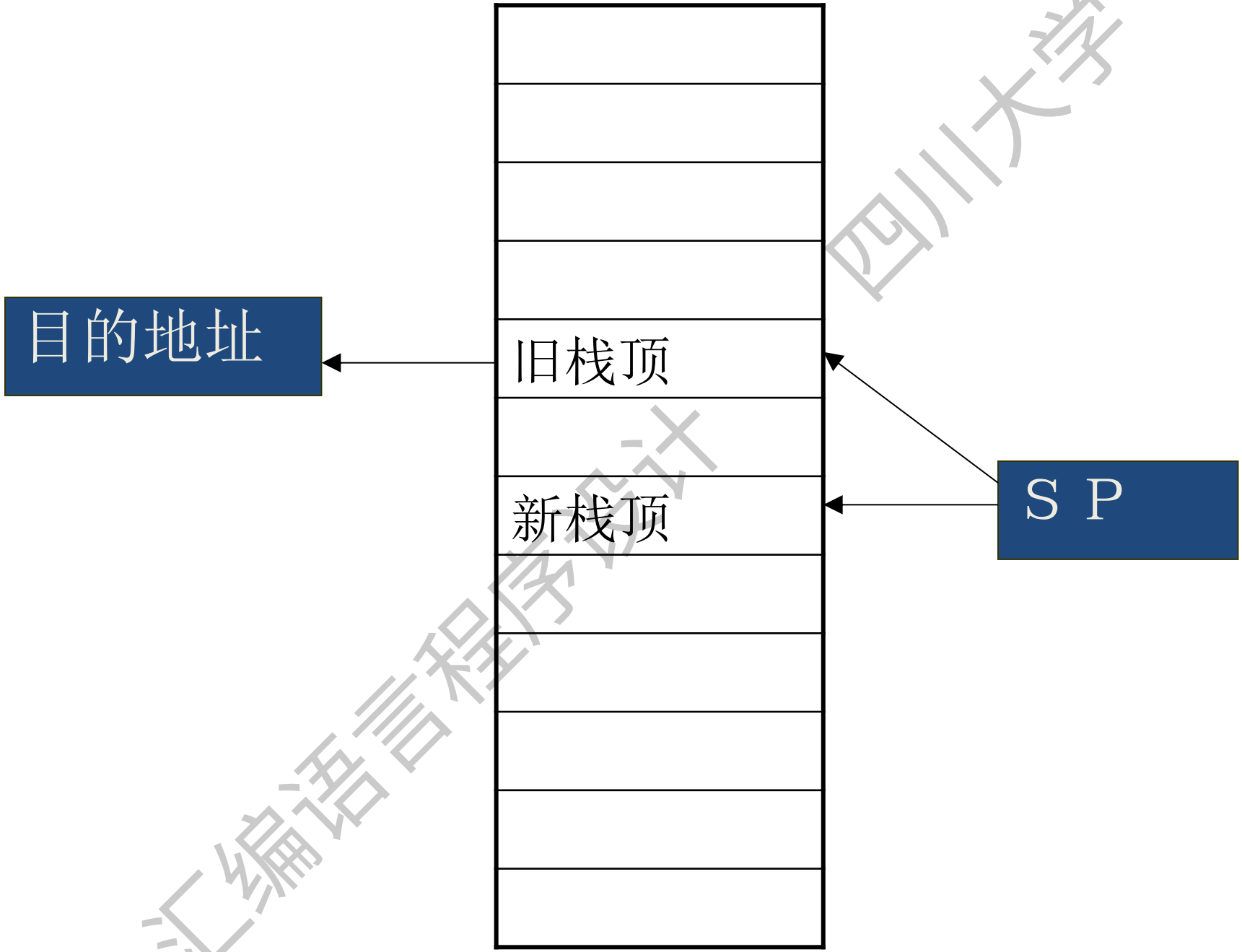
2）出栈指令

- POP指令的执行步骤如下：

- （（SP））-> DEST
- 把SP当前指向的栈顶数据送到指定地址保存

- （SP）+2 -> SP
- 修改栈顶指针，使它向高地址移动一个字单元

- 如果出栈数据要保存到FR中，可以使用POPF指令，隐含的把FR作为操作数地址。



堆栈的组织

- 堆栈段初始化后，SS寄存器指示堆栈段的段基址，即该段的最低地址，SP则指向堆栈段的栈底+2的位置。

- 堆栈段的最高地址-1称为堆栈的**栈底**，SP所指向的地址称为堆栈的**栈顶**。

堆栈的组织

- 堆栈中存放的是**字数据**，16位数据的低8位存放在低地址单元，高8位存放在高地址单元。

- 在程序中定义一个堆栈段的格式如下：

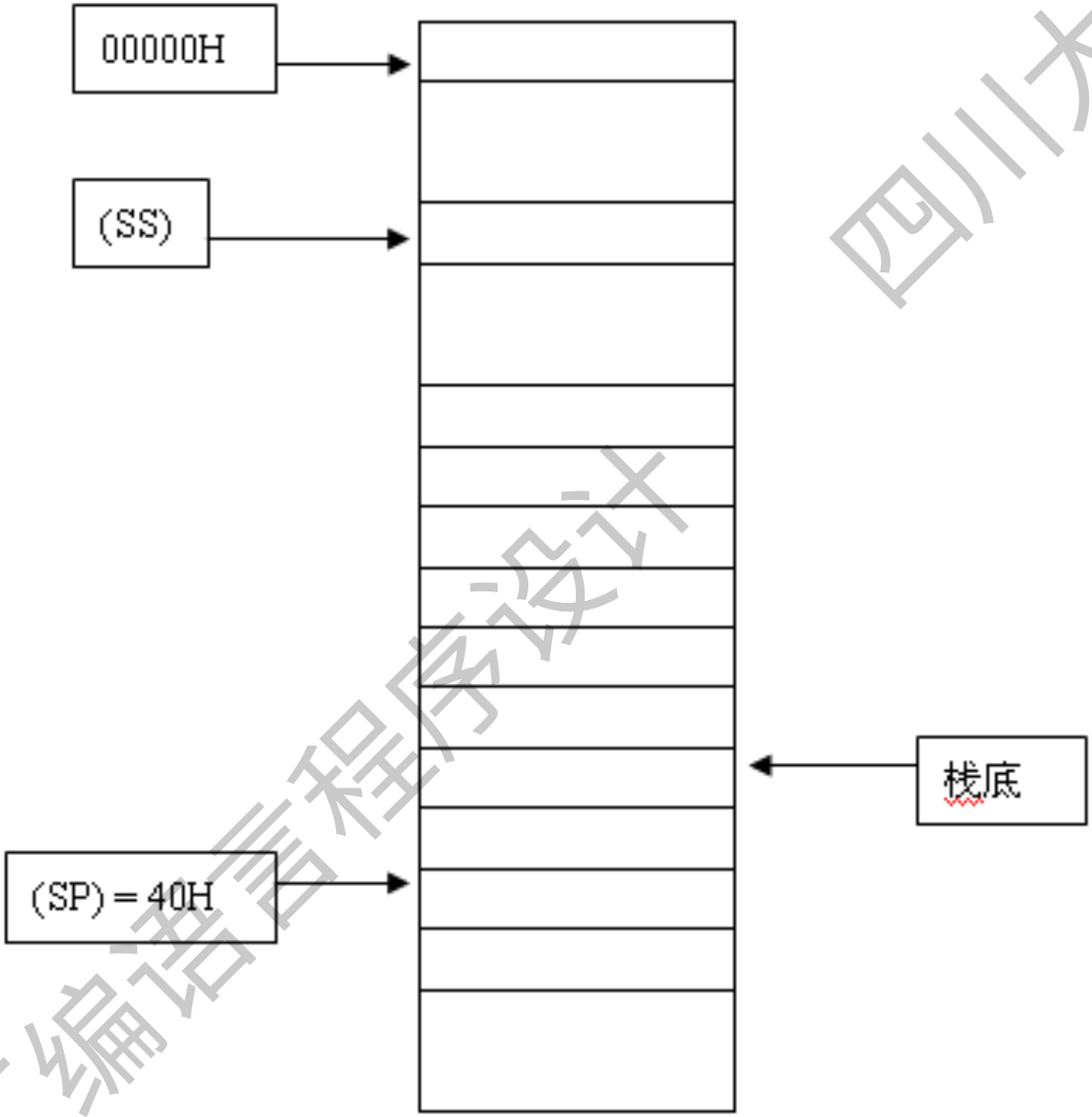
- STACK1 SEGMENT STACK
- DB 40H DUP（0）
- STACK1 ENDS

堆栈的组织

- 这个例子中定义了一个在程序中名称为STACK1的堆栈段，
- 第一行的SEGMENT用于向汇编程序说明现在是在定义一个段，STACK关键字说明这个段的性质是堆栈段；
- DB是以字节为单位分配空间，第二行分配40H个字节作为堆栈段STACK1的空间；

堆栈的组织

- 第三行用ENDS关键字来结束一个段的定义，注意SEGMENT和ENDS关键字前必须使用相同段名称，否则会出现语法错误。



堆栈的组织

- **完成初始化**时，栈顶地址高于栈底地址，因为堆栈中无数据，是一个**空栈**。
- 每次入栈时，SP的内容自动减2，逐步向SS指向的地址移动。
- SS所指向的地址是堆栈的上限地址，如果SP指向的栈顶越过这个地址，则称为**堆栈溢出**。
- 为了避免堆栈溢出，程序员一定要充分估计程序使用的最大堆栈空间，在设置堆栈段时将空间留够。

堆栈操作

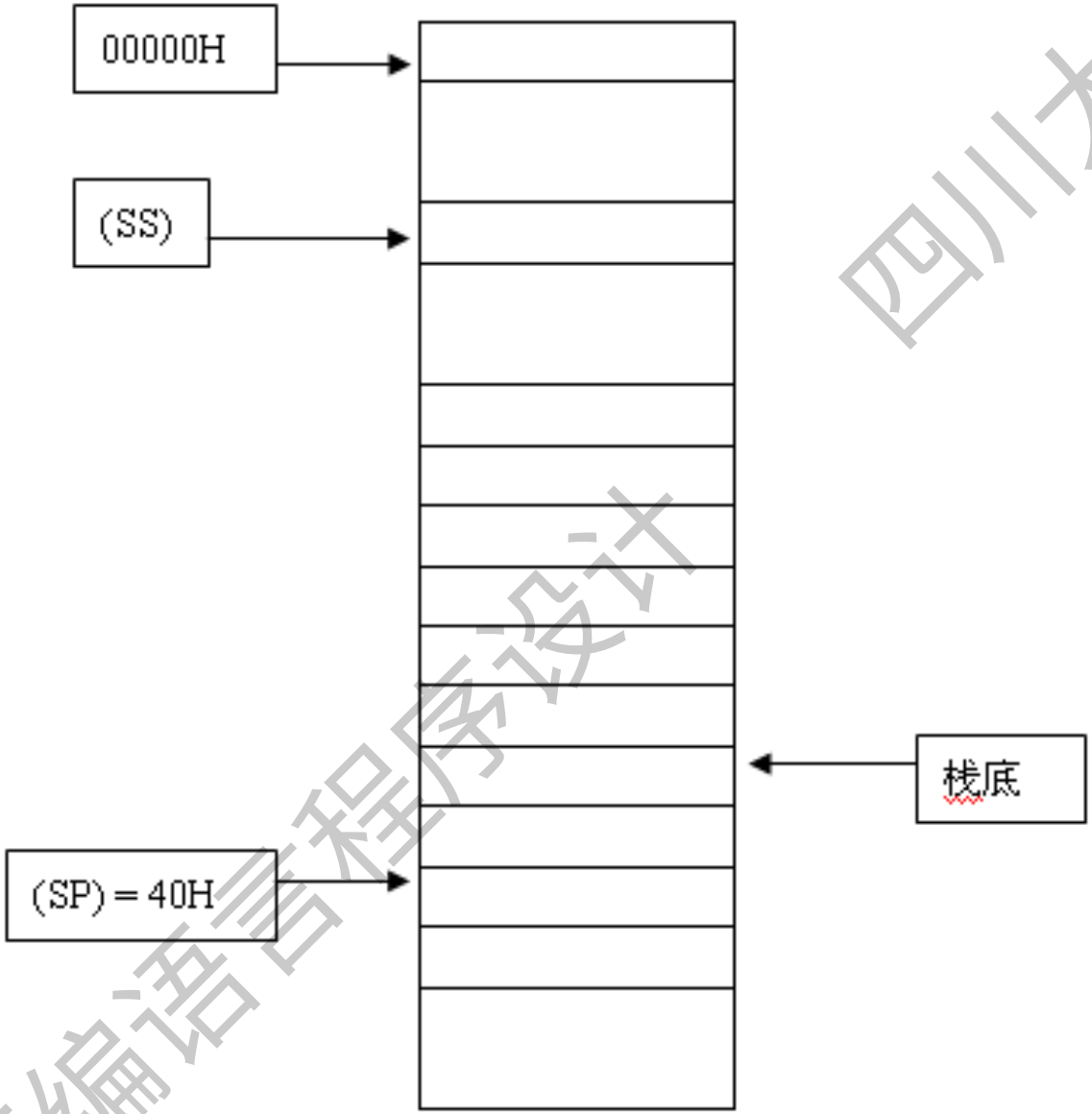
- 例3．通过一段由堆栈指令组成的程序片断来详细说明堆栈的操作过程。

- …
- STACK1 SEGMENT STACK
- DB 40H DUP（0） ；堆栈空间为32个字
- STACK1 ENDS
- …

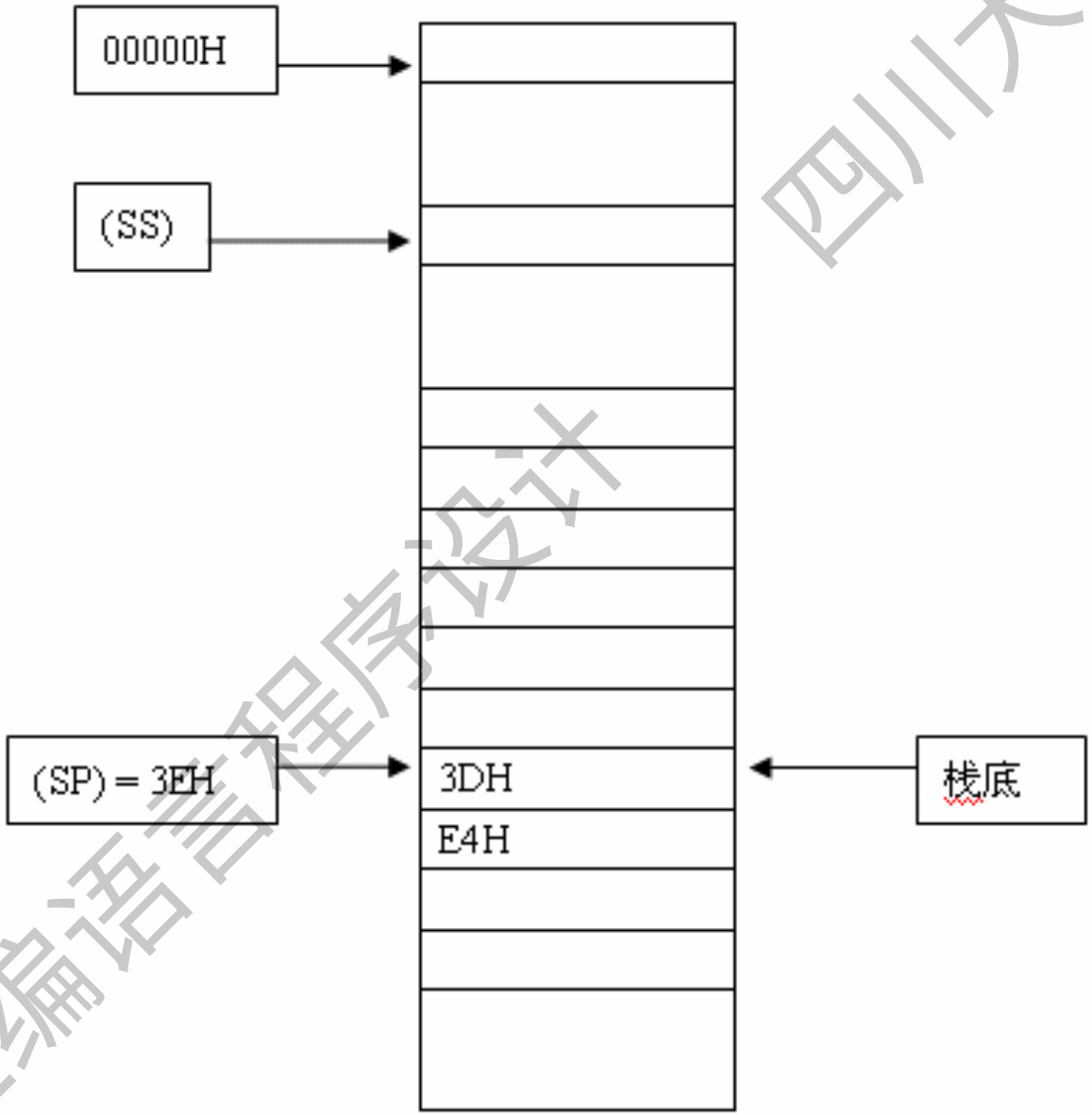
堆栈操作

- PUSH AX ；假定（AX）=0E43DH
- PUSH DS ；假定（DS）=3638H
- PUSH DATAW ；假定（DATAW）=0D245H
- PUSHF ；假定（FR）=0C243H
- POPF
- POP DATAW
- POP DS
- POP AX
- …

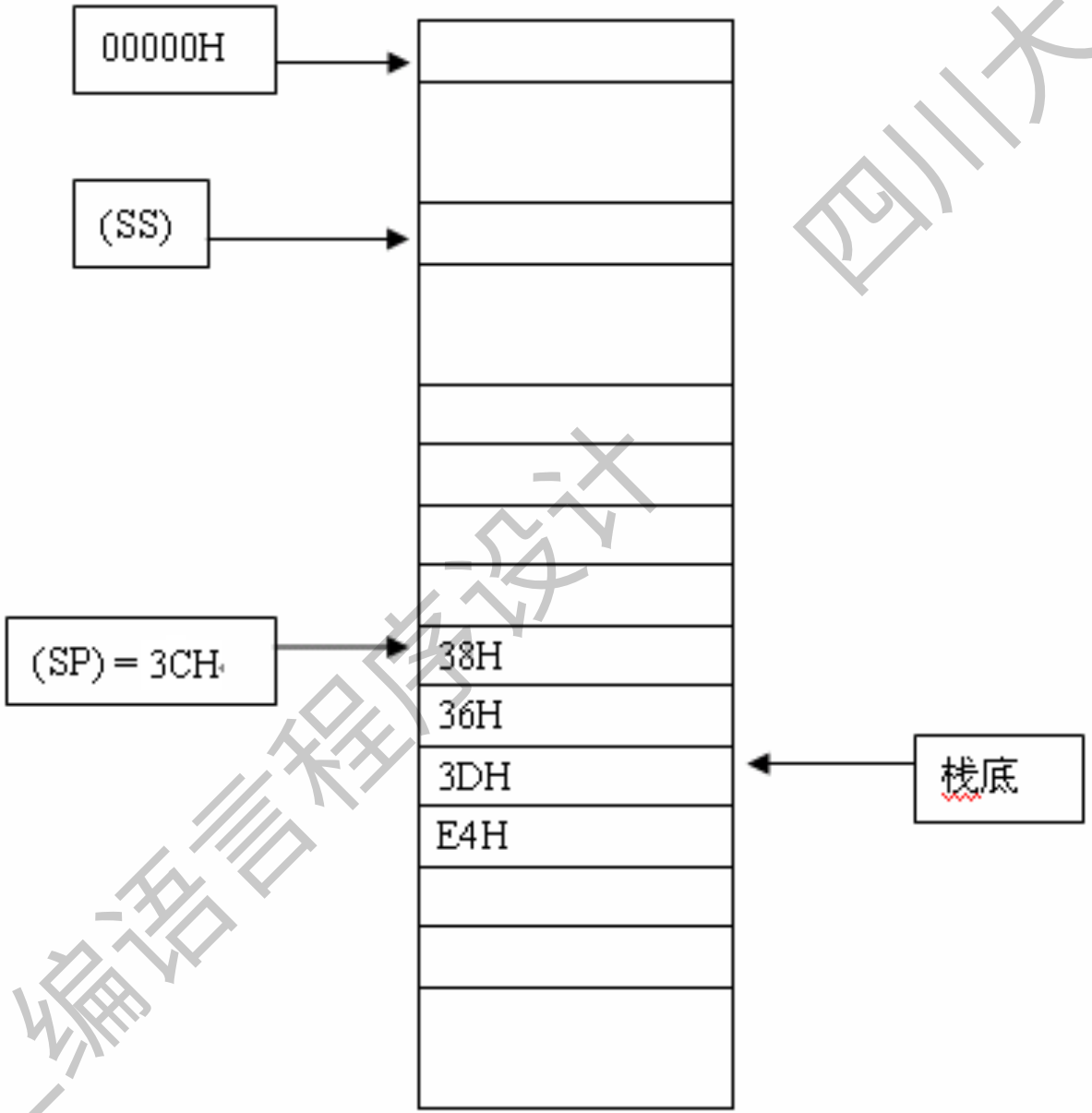
- 堆栈段STACK1初始化以后情况如下所示：



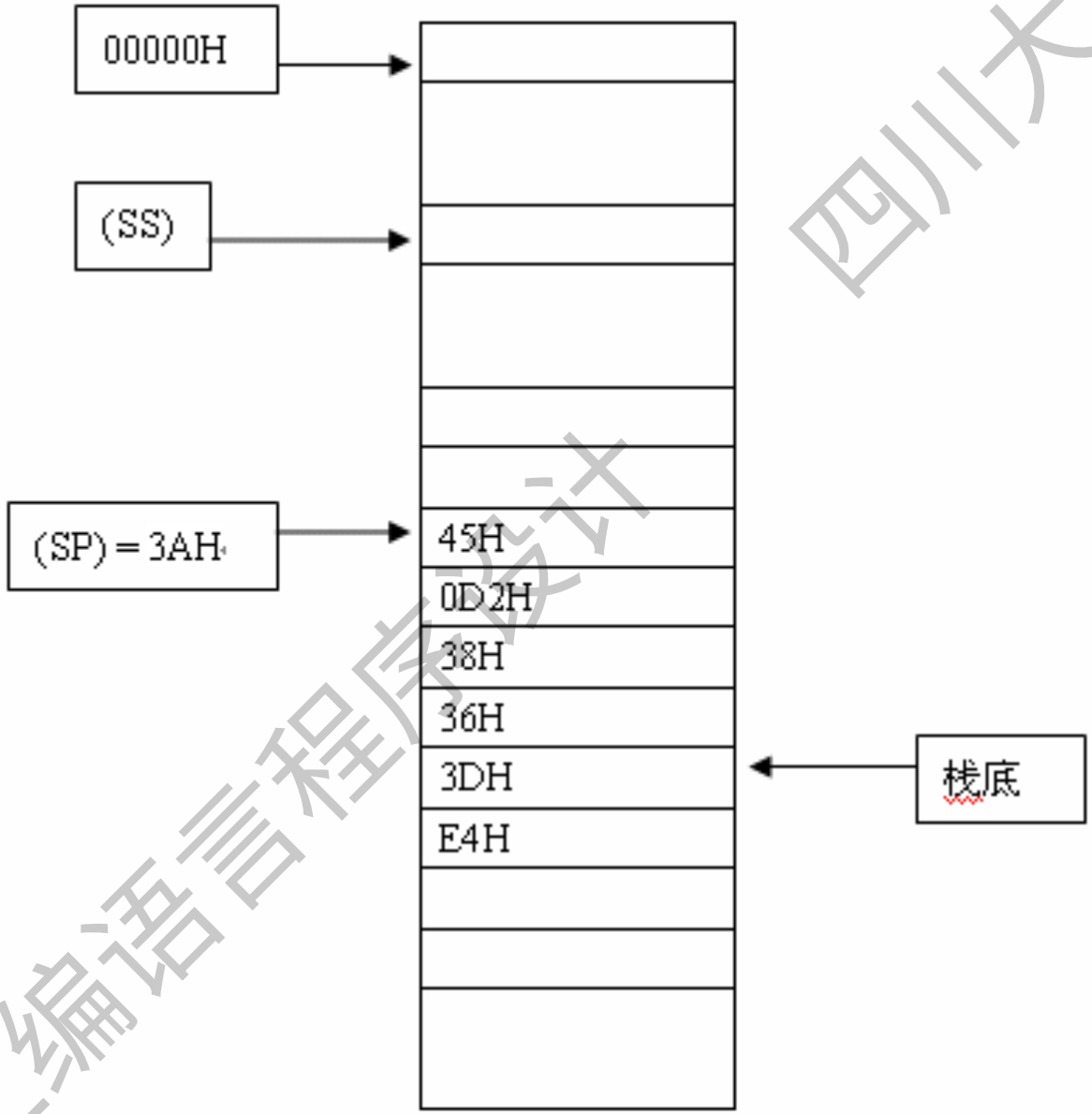
（AX）=0E43DH，执行 PUSH　AX 后，堆栈变为如下情况：



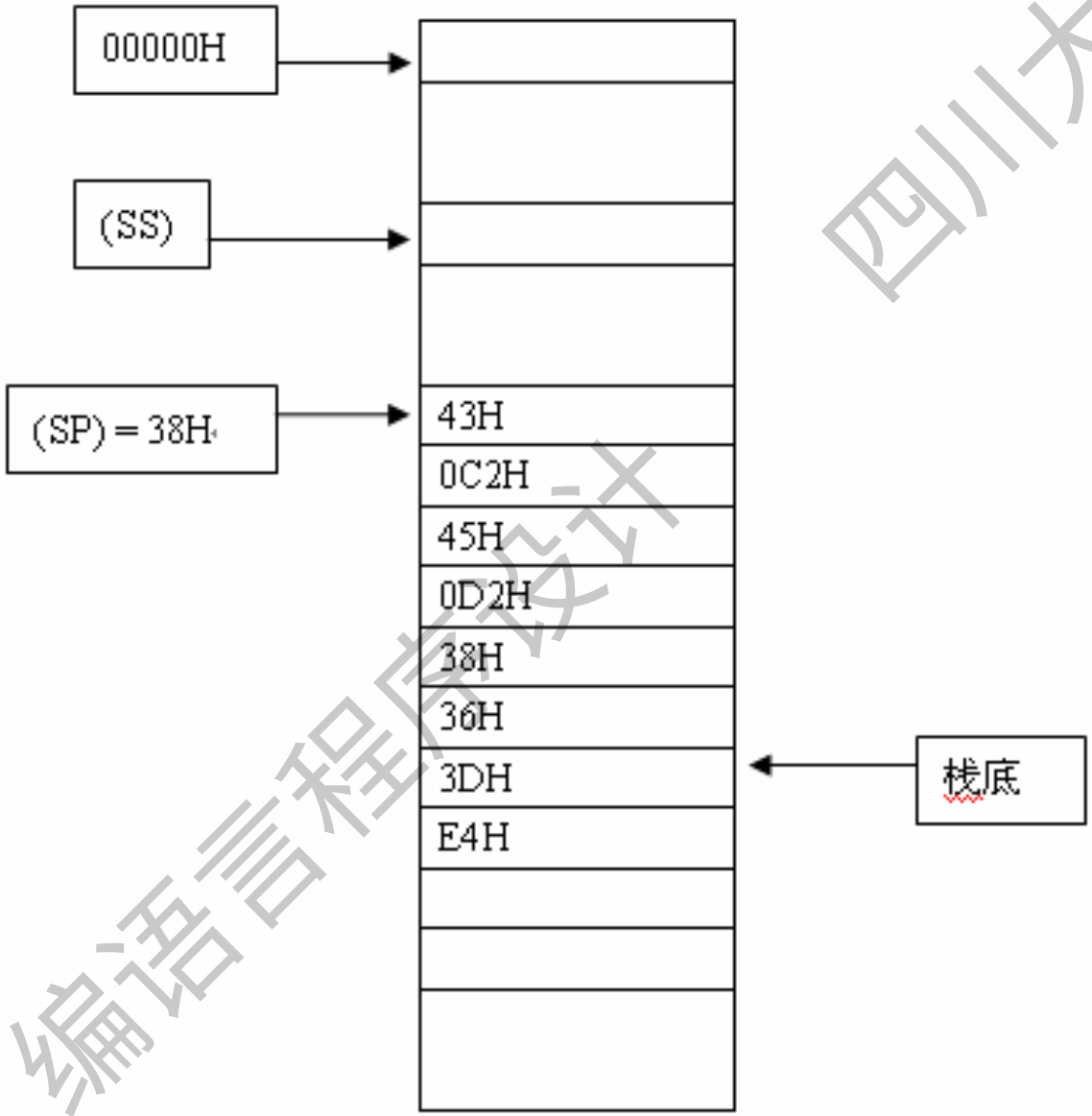
（DS）=3638H，执行 PUSH　DS 后，堆栈变化情况如下：



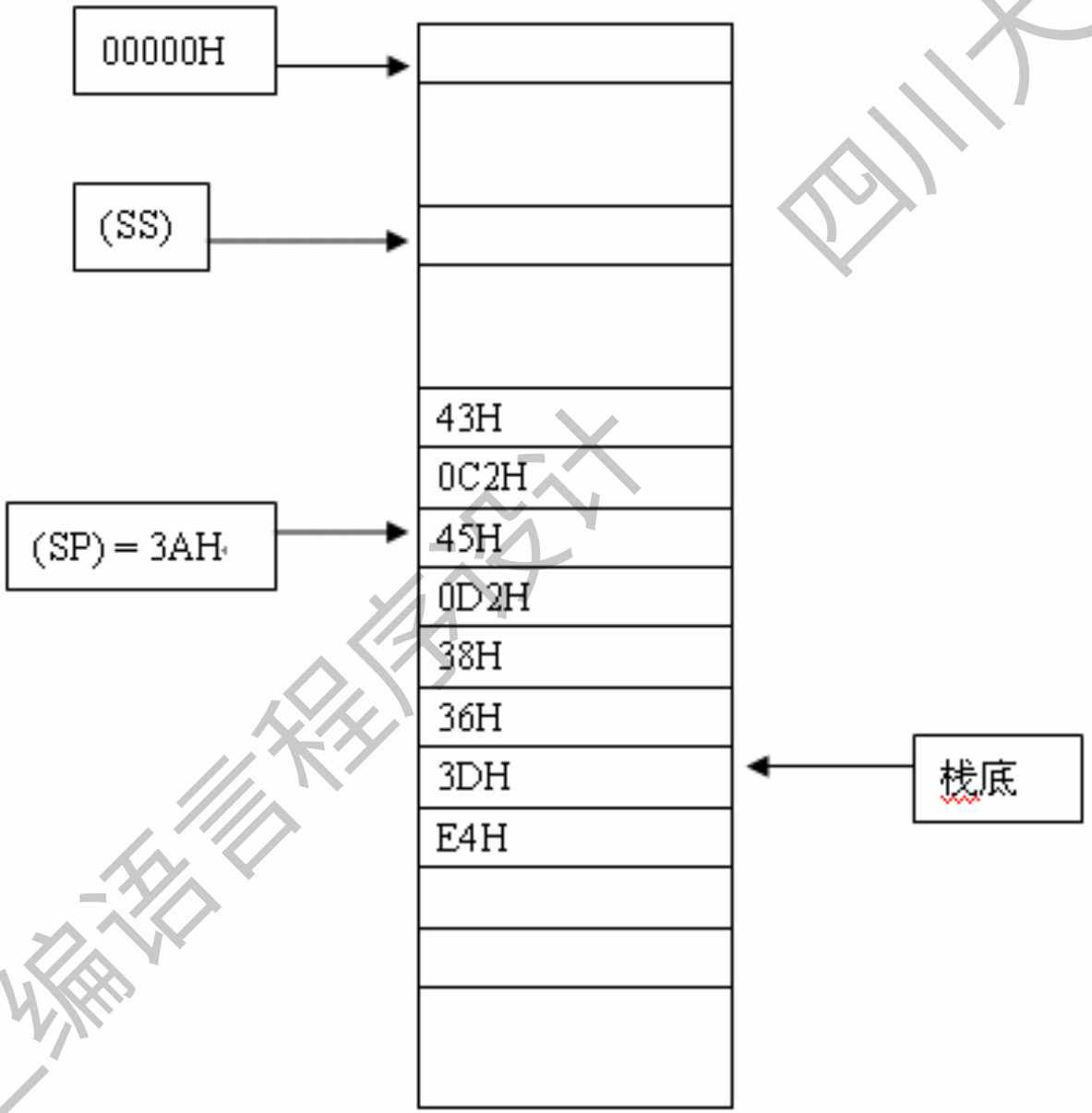
（DATAW）=0D245H，执行 PUSH　DATAW 后，堆栈变化情况如下：



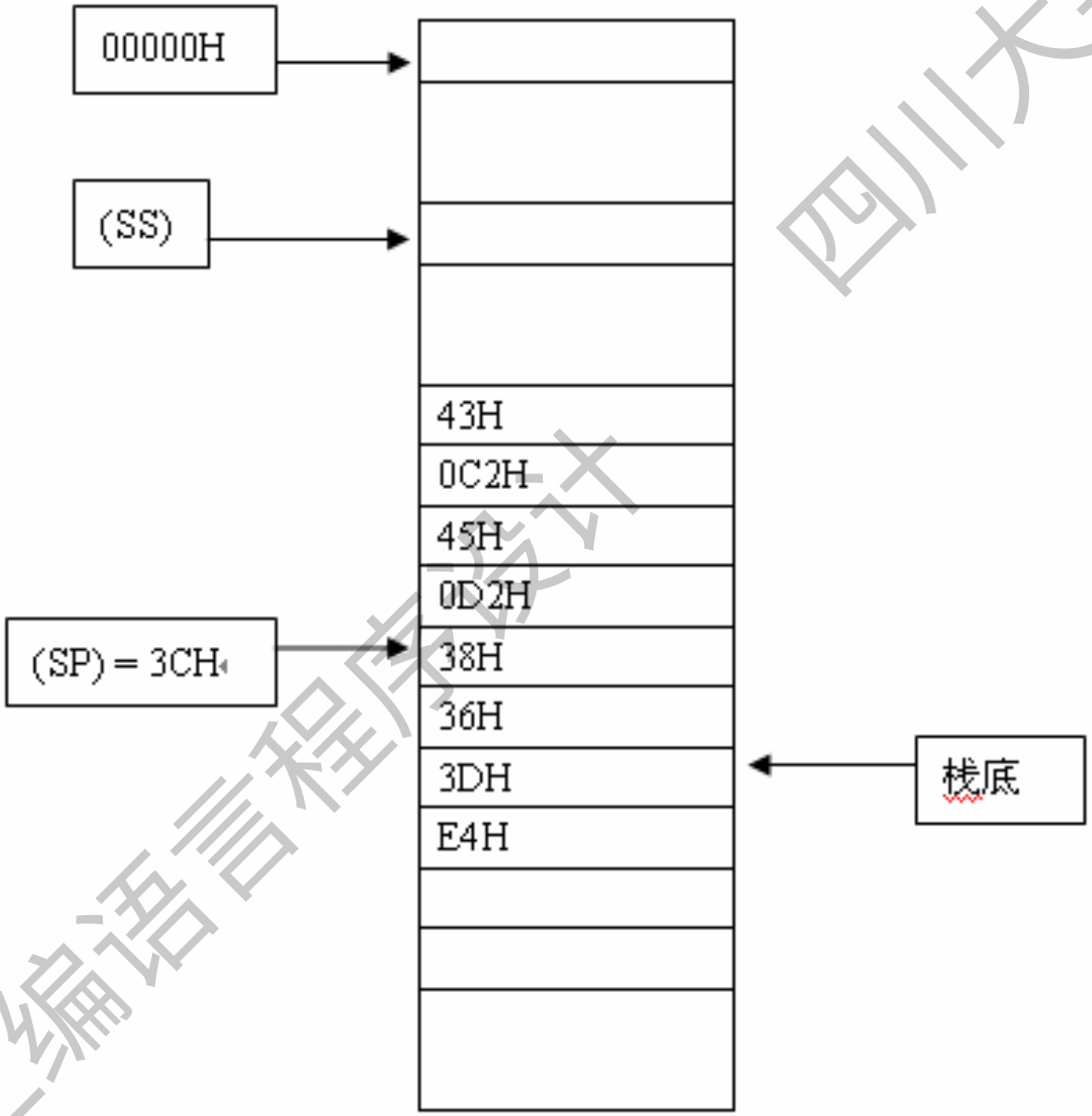
（FR）=0C243H，执行 PUSHF 后，堆栈变为如下情况：



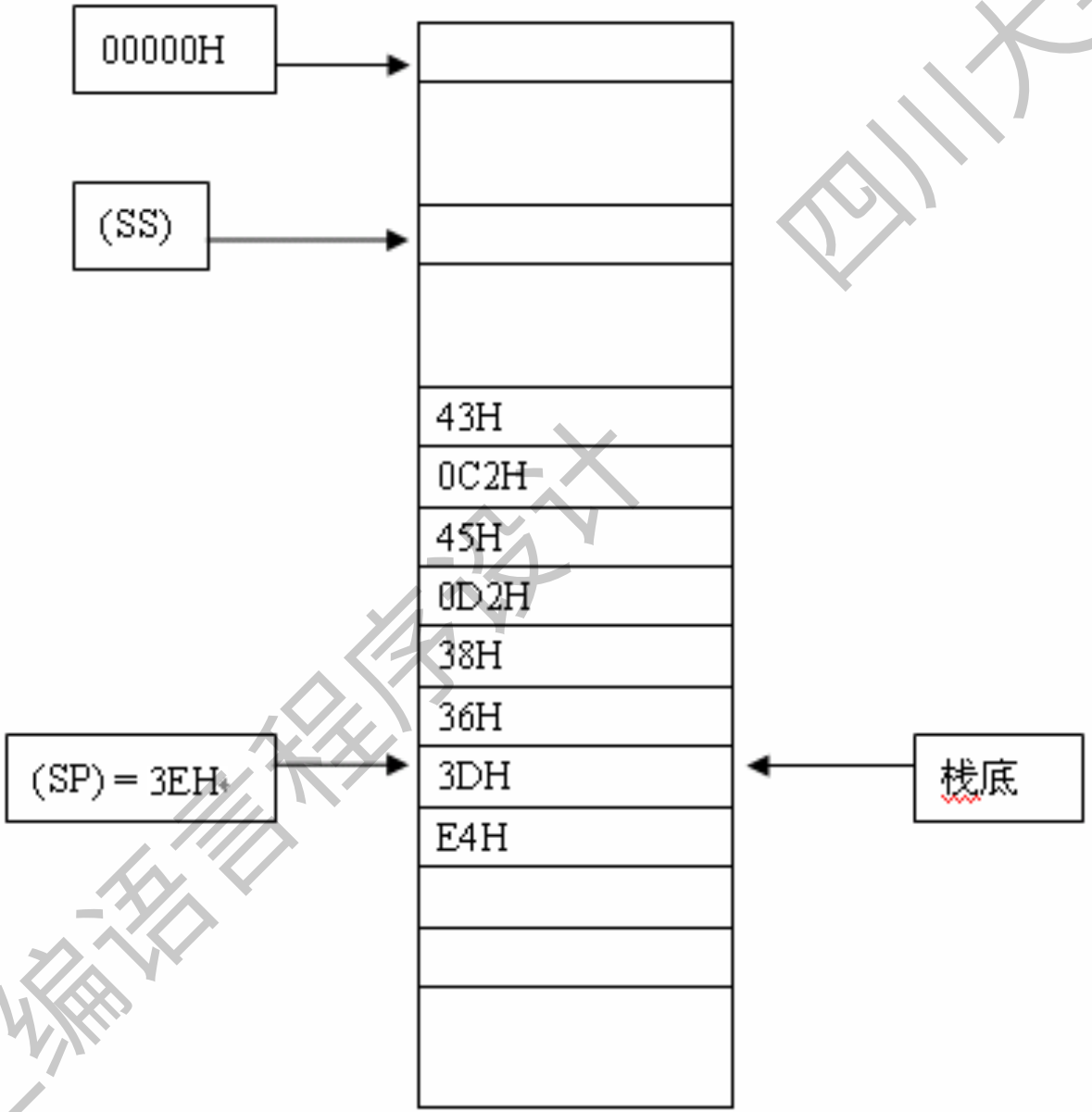
执行 POPF 后，堆栈的变化情况如下：



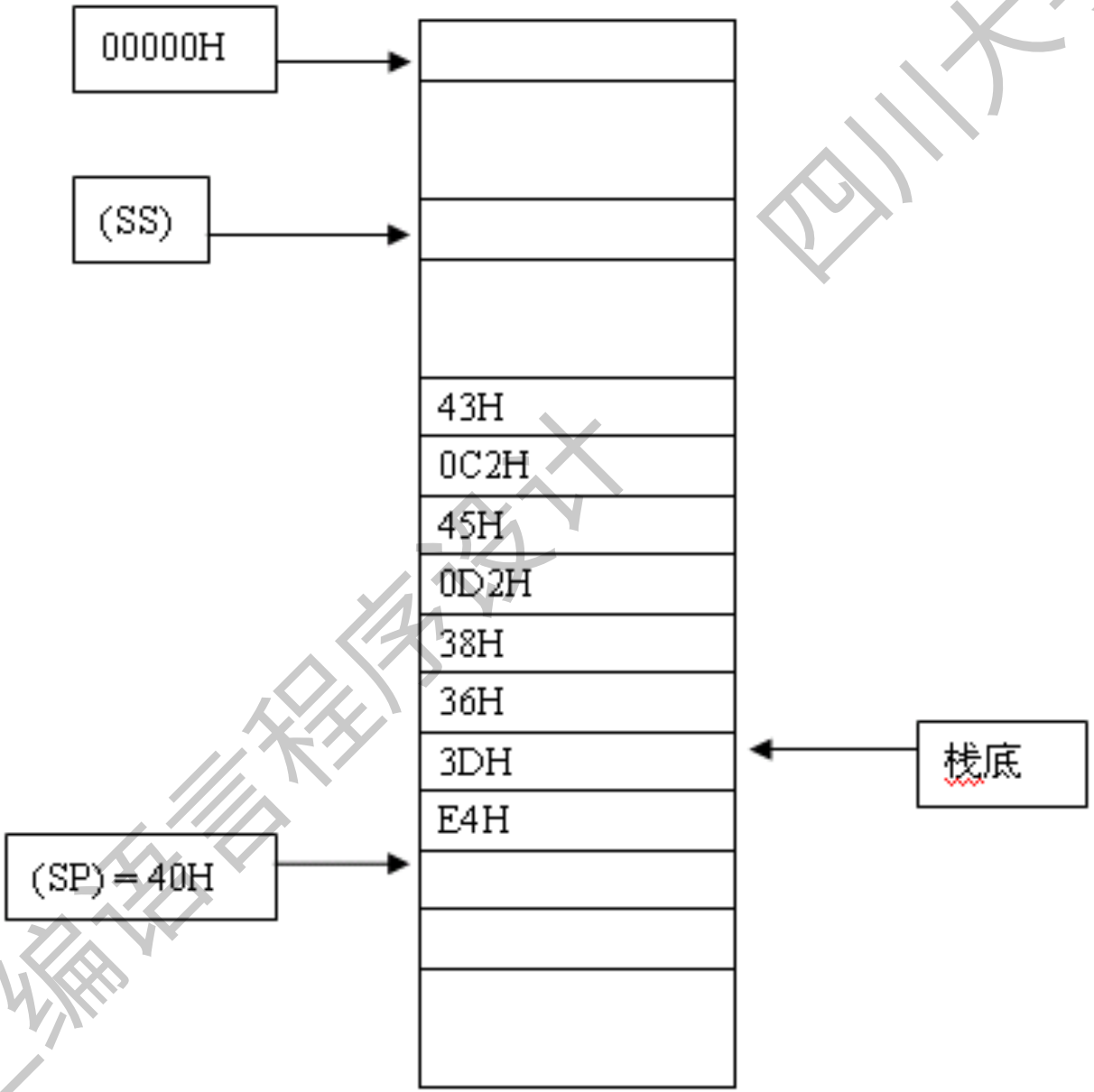
执行 POP　DATAW 后，堆栈变化情况如下：



执行 POP　DS 后，堆栈变化情况如下：



执行 POP　AX 后，堆栈的变化情况如下：



堆栈操作

- 数据**出栈顺序一定要和入栈顺序相反**，否则不能恢复正确的数据。

（4）标志位传送指令

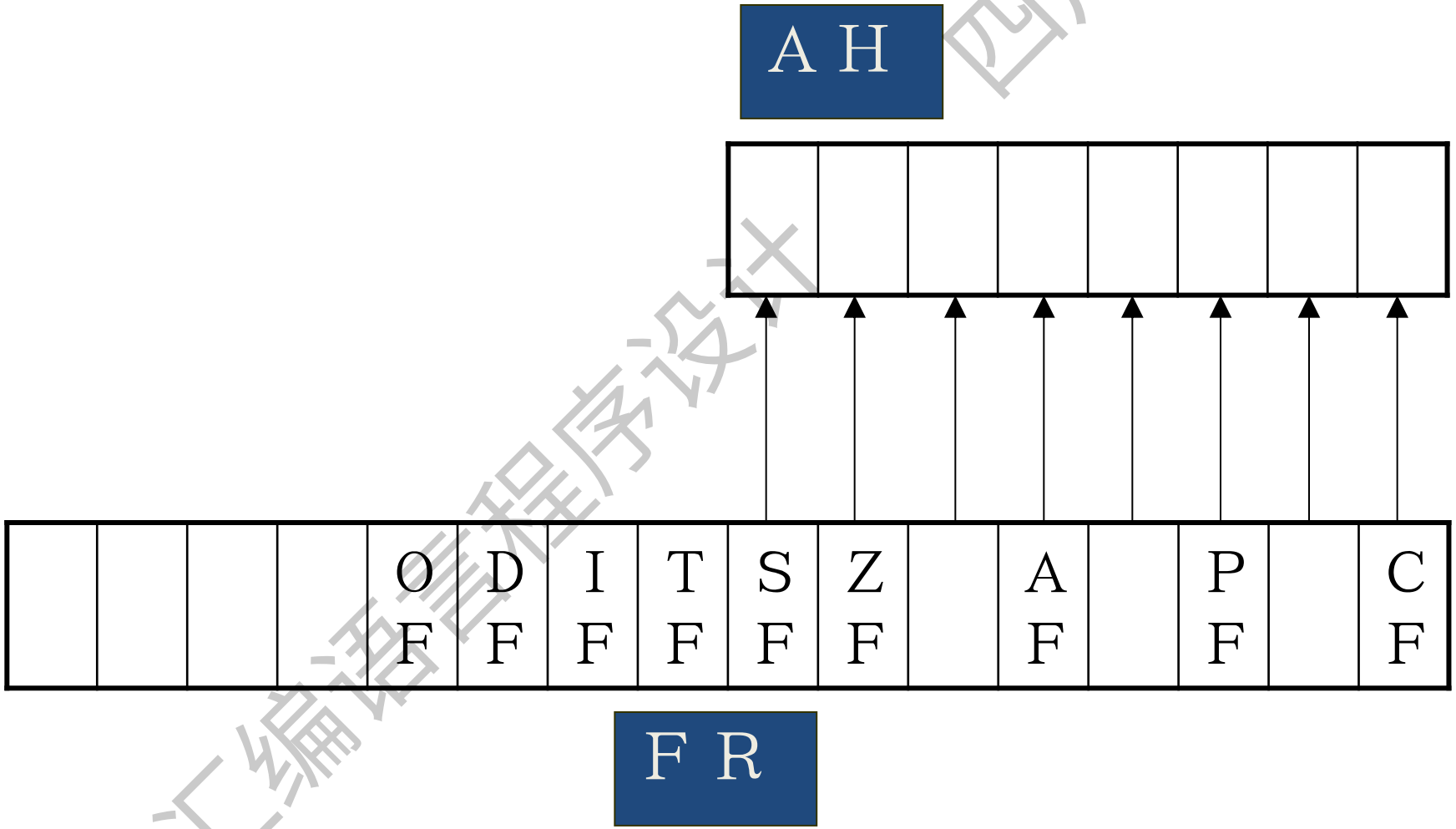
- 1）取标志指令（Load register AH from flag）
- 指令格式：LAHF

- 无操作数指令，使用了两个隐含操作数，由机器指令的OPCODE指定。

- 指令功能：AH <=（FR）7-0，

- 标志位影响：无

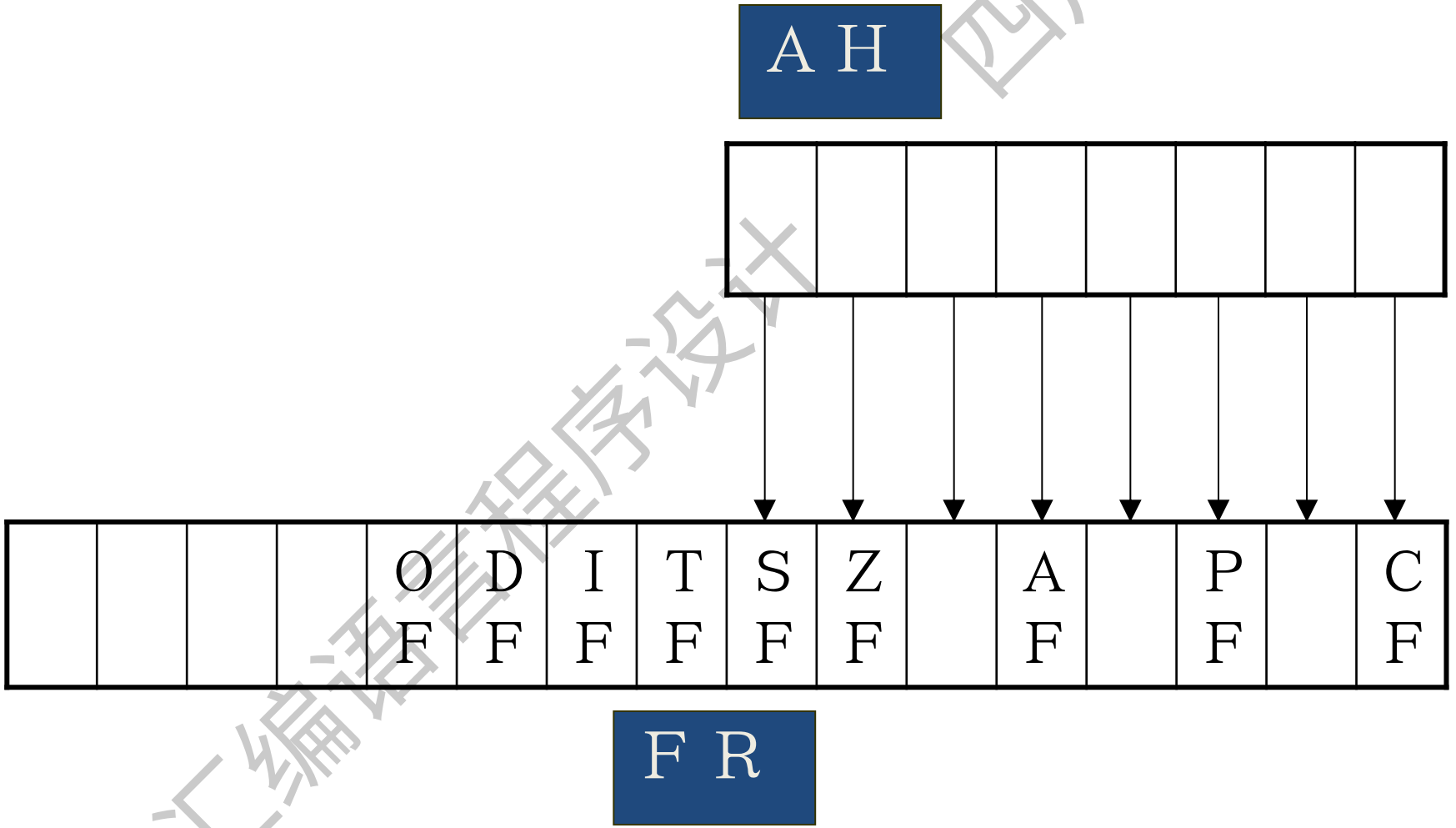
（4）标志位传送指令



（4）标志位传送指令

- 2）存标志指令（Store register AH into flag）
- 指令格式：SAHF
- 无操作数指令，使用两个隐含的寄存器操作数。
- 指令功能：FR7-0 <=（AH）
- 标志位影响：SF、ZF、AF、PF、CF

（4）标志位传送指令



LAHF 与 SAHF的应用

- CMP BYTE PTR [BX], 02H
- LAHF ; Protect FR
- INC BX
- SAHF ; Restore FR
- JZ L1 ; FR is from CMP, not INC
- JMP L2
- ……

（4）标志位传送指令

- 3）标志压栈指令（Push flag）
- 指令格式：PUSHF
- 指令功能：SP <= (SP)-2
- (SP)<= (FR)
- 把标志寄存器中的内容保存到堆栈中。
- 标志位影响：无

（4）标志位传送指令

- 4）标志出栈指令（Pop flag）
- 指令格式：POPF
- 指令功能：FR<= ((SP))
- SP<= (SP)+2
- 从栈顶取数据保存到标志寄存器。
- 标志位影响：所有标志位

PUSHF 与 POPF的应用

- CMP BYTE PTR [BX], 02H
- PUSHF　　; Protect FR
- INC　BX
- POPF　　; Restore FR
- JG　　L1　; FR is from CMP, not INC
- JMP　L2
- ……

（5）地址传送指令（Address transfer）

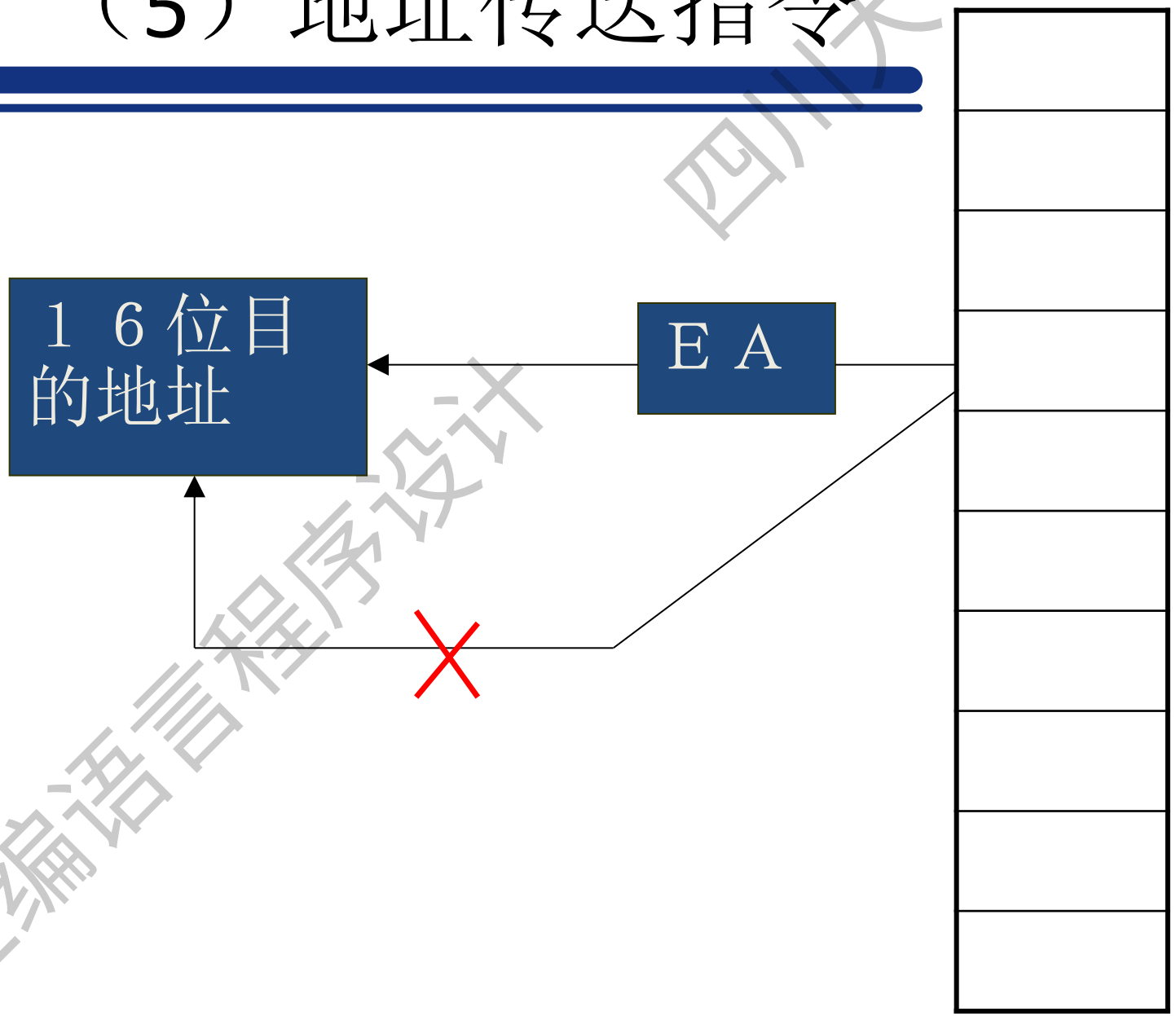
- 1）装入有效地址指令（Load Effective Address）
- 指令格式：LEA DEST， SRC
- 指令功能：DEST <= SRC的EA
- 将源操作数的有效地址送到目的地址中保存。
- 标志位影响：无

（5）地址传送指令

- LEA指令的源操作数必须是内存操作数，只有内存操作数的寻址方式中有有效地址这一概念

- 目的操作数只能是16位的通用寄存器，因为有效地址是一个16位的无符号二进制编码。

（5）地址传送指令



（5）地址传送指令

- LEA指令原理示例：

- MOV BX，0024H
- LEA　BX，[BX]

- 假定逻辑地址（DS）：0024H指向的内存单元内容为0056H

- 执行上述两条指令后，（BX）=?

（5）地址传送指令

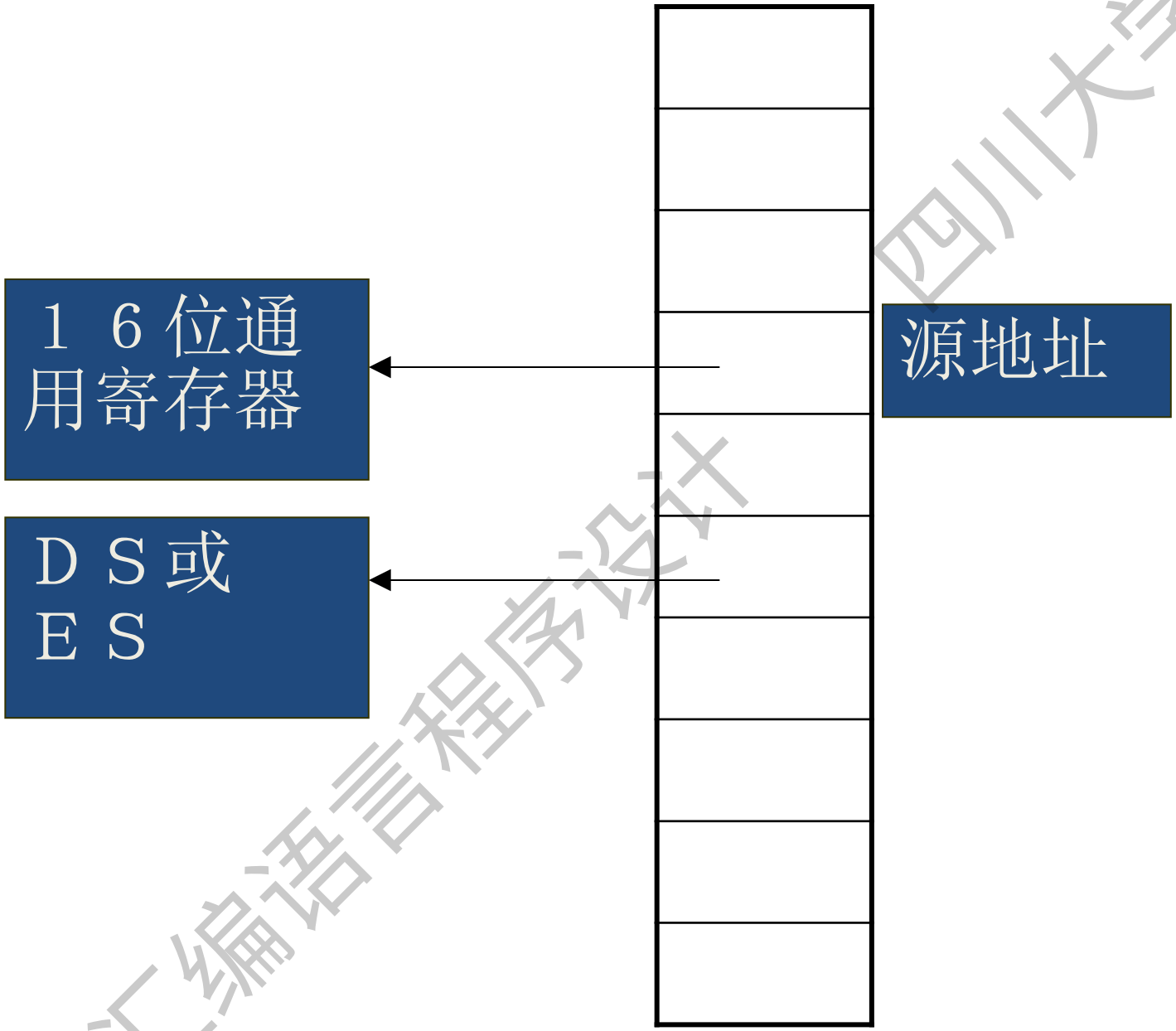
- LEA指令原理示例：
- LEA　SI，DS：[0056H]
- 假定逻辑地址（DS）：0056H指向的内存单元内容为0048H。
- 执行上述指令后，（SI）=?

（5）地址传送指令

- 2）装入地址指针指令
- 指令格式：LDS DEST， SRC
- LES DEST， SRC

- 指令功能：
- LDS：DEST <=（SRC）
- DS <=（SRC+2）

- LES：DEST <=（SRC）
- ES <=（SRC+2）



（5）地址传送指令

- 这两条指令假定源操作数地址SRC中存放着2个字构成的逻辑地址
- 低地址的字被解释为逻辑地址中的有效地址
- 高地址的字被解释为逻辑地址中的段基值

（5）地址传送指令

- 指令执行时把低字内容传送到目的地址（只能是16位的通用寄存器），把高字内容传送到DS（ES）段寄存器。

- 概括说来，就是传送一个完整的逻辑地址，包括偏移量和段基值。

- 标志位影响： 无

（5）地址传送指令

- LDS指令原理示例：
- MOV　BX，0024H
- LDS　BX，[BX]
- MOV　AX，[BX]

- 假定初始值：
- （DS）=02F8H
- （02F8H：0024H）= 0056H
- （02F8H：0026H）= 06A4H
- （02F8H：0056H）= 00B2H
- （06A4H：0056H）= 0008H
- 执行上述指令后，（DS）=？　（BX）=？　（AX）=？

（5）地址传送指令

- LDS、LES两条指令常用于处理不在当前数据段和附加段的操作数。

- 在串操作指令中，隐含使用DS：SI指向源串，ES：DI指向目的串。

（5）地址传送指令

- 如程序中需处理的串不在当前数据段或附加段中，就必须修改DS、ES。

- 这时可用LDS与LES指令来实现，并且在指令中使用SI、DI作目的地址，让SI、DI分别存放源串与目的串的偏移量。

LEA与LDS、LES的区别

- ＬＥＡ指令把内存单元的ＥＡ计算出来并存放到１６位通用寄存器中，和内存单元中的内容无关.

- ＬＤＳ（ＬＥＳ）指令把内存单元中的内容存放到１６位通用寄存器和段寄存器，把内存单元中的内容解释为完整的逻辑地址，和内存单元实际的ＥＡ与段基值没有任何关系.

第三章 8086/8088寻址方式及指令集(算术运算指令)

2.算术运算类指令

- 完成加、减、乘、除等二进制算术运算。
- 对于加减法，无符号数与补码的运算使用相同的指令。
- 对于乘除法，无符号数与补码的运算使用不同的指令。

（1）加法指令

- 指令格式：ADD DEST， SRC
- 指令功能：DEST <=（SRC）+（DEST）

- 标志位影响：OF、SF、ZF、AF、PF、CF

- 具体哪些标志位有意义依赖于用户对操作数的解释。

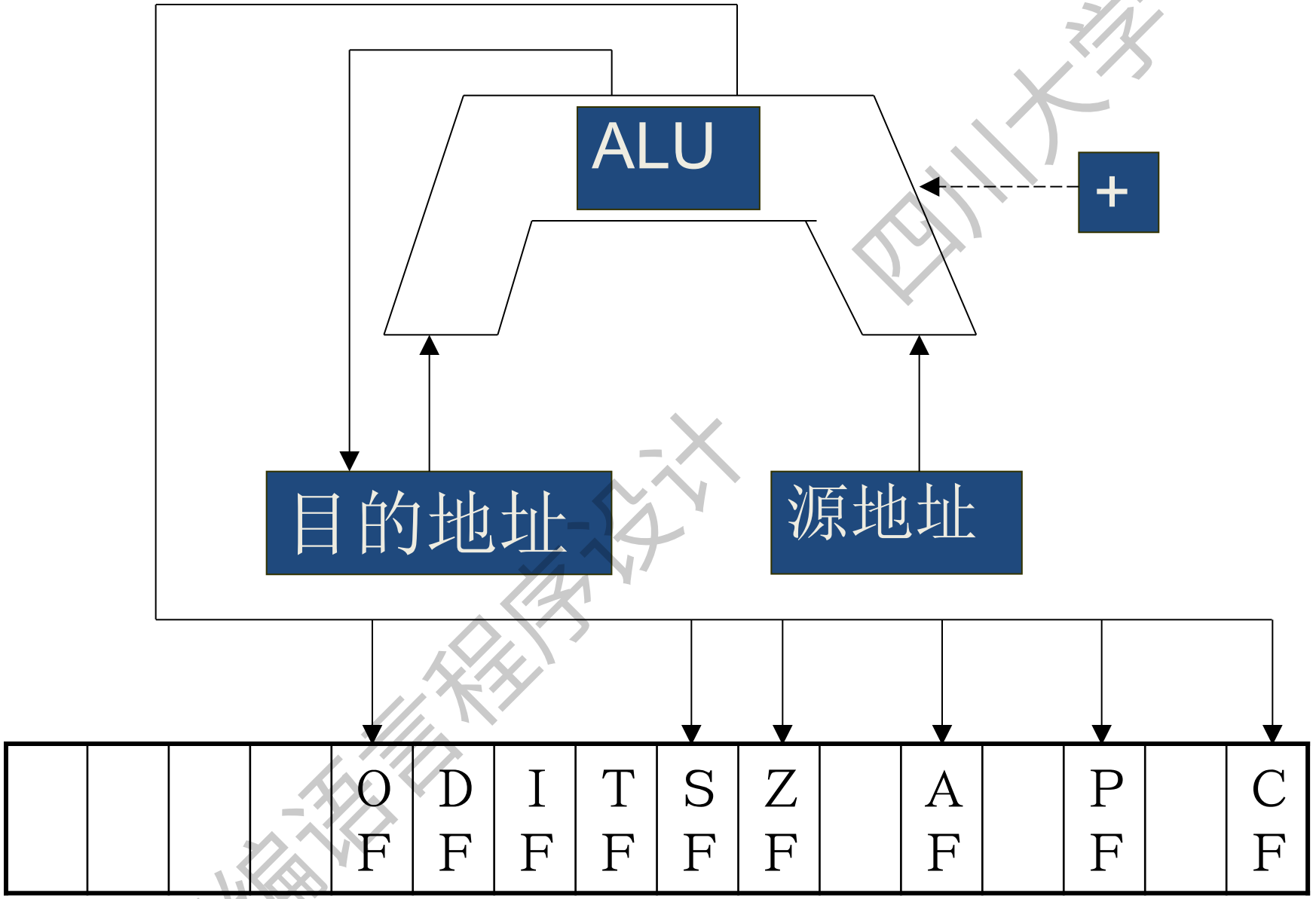
（1）加法指令

- 操作数可以字、字节为单位；
- 源操作数可以为通用寄存器、内存单元、立即数；
- 目的操作数可以为通用寄存器、内存单元。
- 注意：最多只能有一个内存操作数。

（1）加法指令

- 加法指令的通常格式有：

- ADD BX，SI
- ADD DA_WORD,0F8CH
- ADD DL,TAB[BX]



（1）加法指令

- 例1.以下面的程序片段为例说明加法指令的执行情况。

- MOV AH，45H
- MOV AL，0E3H
- ADD AH，AL

- 45H = 01000101B
- 0E3H =11100011B

- 如果看作无符号数相加，是完成69+227，如果看作带符号数相加，是完成69+（-29）。

（1）加法指令

- 01000101
- \+ 11100011
- 1 00101000

- 看作无符号数相加：最高位进位，CF=1
- 第3位没有产生进位，AF=0
- 看作带符号数相加：正+负 OF=0，SF=0
- 运算结果为非0，ZF=0；运算结果中1的个数为偶数个，PF=1

- 课下请使用DEBUG来观察指令执行情况。

（2）带进位加法指令

- 指令格式：ADC DEST， SRC
- 指令功能：DEST <=（SRC）+（DEST）+（CF）
- 标志位影响：OF、SF、ZF、AF、PF、CF
- 具体哪些标志位有意义决定于用户对操作数的解释。

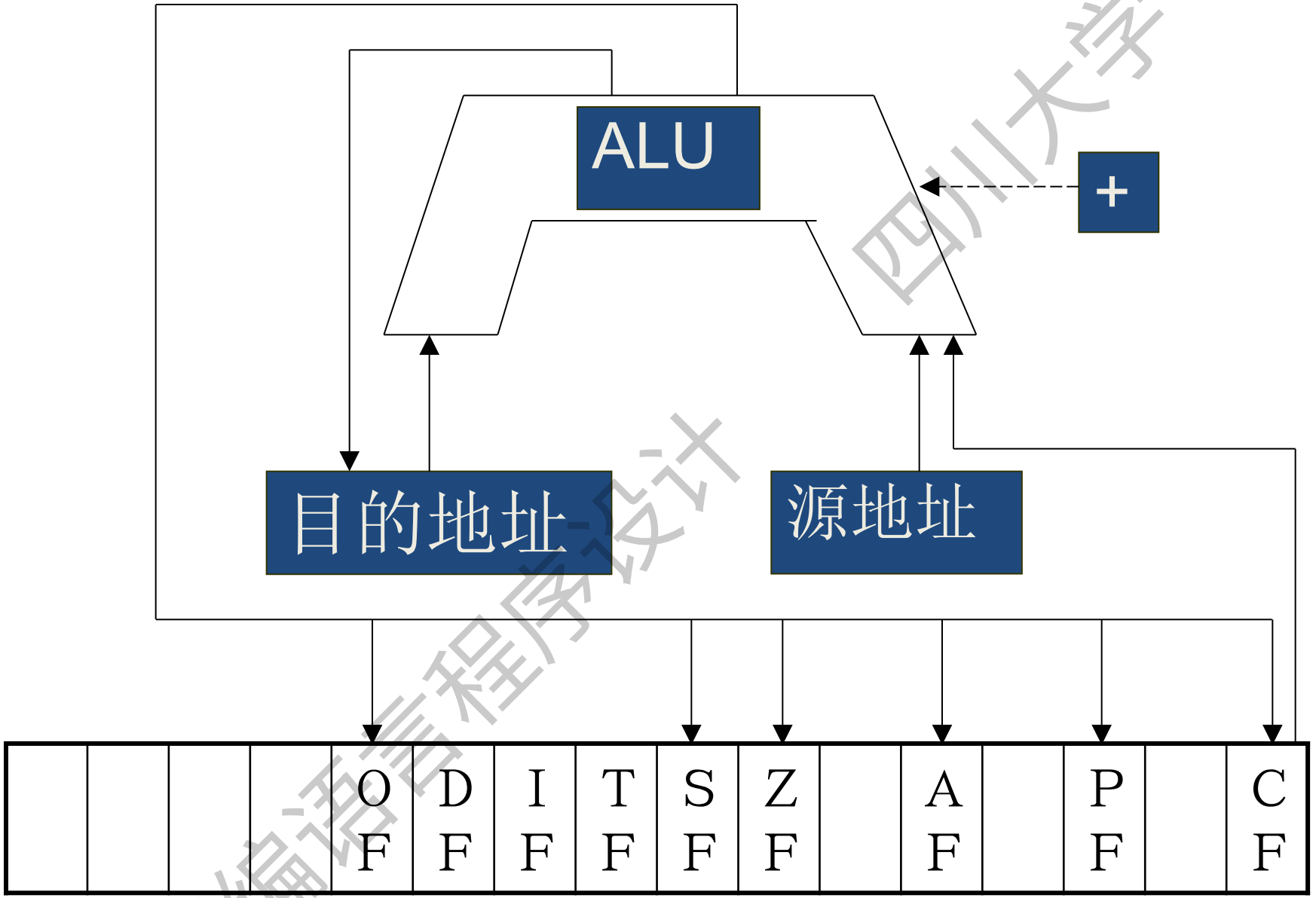
（2）带进位加法指令

- 操作数可以字、字节为单位；
- 源操作数可以为通用寄存器、内存单元、立即数；
- 目的操作数可以为通用寄存器、内存单元。
- 注意：最多只能有一个内存操作数。

（2）带进位加法指令

- 例 有一个32位无符号数存放在DX（高16位），AX（低16位）中，若要加上常数76F1A23H，则用以下指令来实现：

- ADD AX，1A23H
- ADC DX，76FH



（2）带进位加法指令

- 例2.用下面的程序片段来说明带进位加法指令的原理。

- MOV AL，93H
- MOV AH，02H
- MOV BL，88H
- MOV BH，0EEH
- ADD AL，BL
- ADC AH，BH

- 先考虑把操作数看作无符号数的情况。

- （AX）= 0293H　　659（十进制）
- （B　　（
- 　　
- 　　0000010　　10010011
- \+　　11101110　　10001000
- 　　0　　11110001　　1　　00011011
- 　　CF　　CF

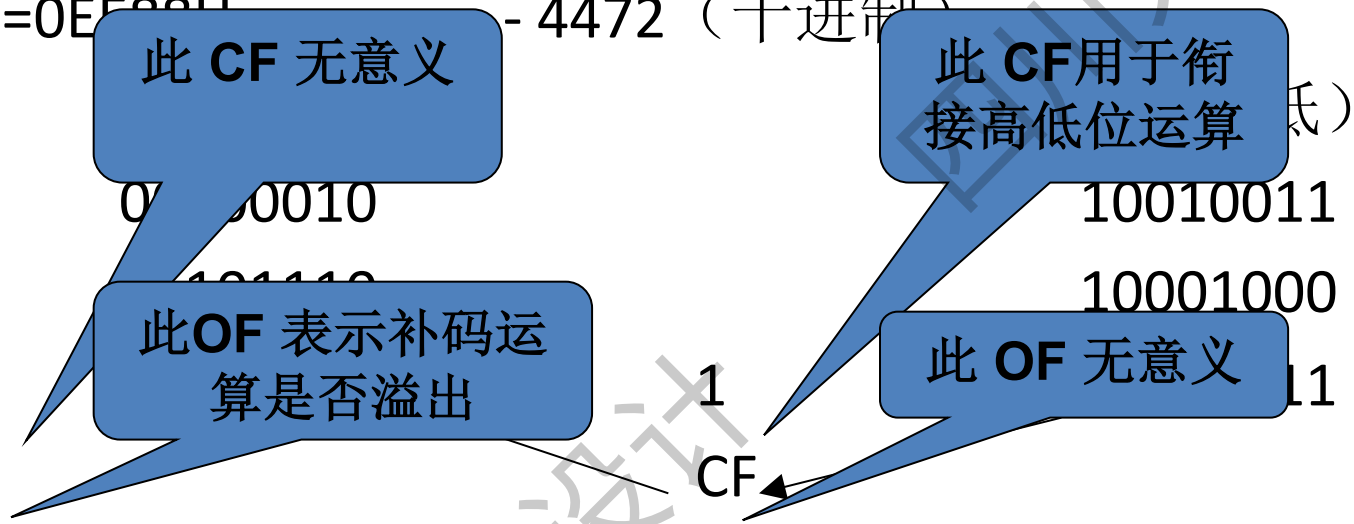
此 CF 意义在于表示无符号数运算是否溢出

此 CF 意义在于衔接高、低位运算

- 运算结果为：F11BH　　61723（十进制）

- 注意，如果最后一次加法后CF=1，那么说明运算结果超出了无符号数的表示范围，需要把CF作为运算结果的最高位才是正确的运算结果。

- 再考虑把操作数看作带符号数的情况。
- （AX）= 0293H　　659（十进制）
- （BX）=0E[illegible]　　- 4472（十进制）
- [illegible]　　[illegible]（补）
- 0[illegible]0010　　10010011
- +　[illegible]　　10001000
- 0　　1[illegible]1
- CF　　CF
- OF=0　　OF=1



- 运算结果为：F11BH　　-3813（十进制）

- 多字节或多字带符号数的加法中，是否溢出仅决定于最后一次加法操作后的OF标志。

（2）带进位加法指令

- 课下请使用DEBUG来观察ADD、ADC指令结合使用实现的多字节加法过程。

- 主要观察CF和OF标志位。

（2）带进位加法指令

- 结合使用ADD、ADC指令可以实现长操作数的加法，摆脱了字和字节的范围限制。

- 如果操作数需要用N个字来表示，那么可以在程序的循环结构中使用ADC指令来完成加法。

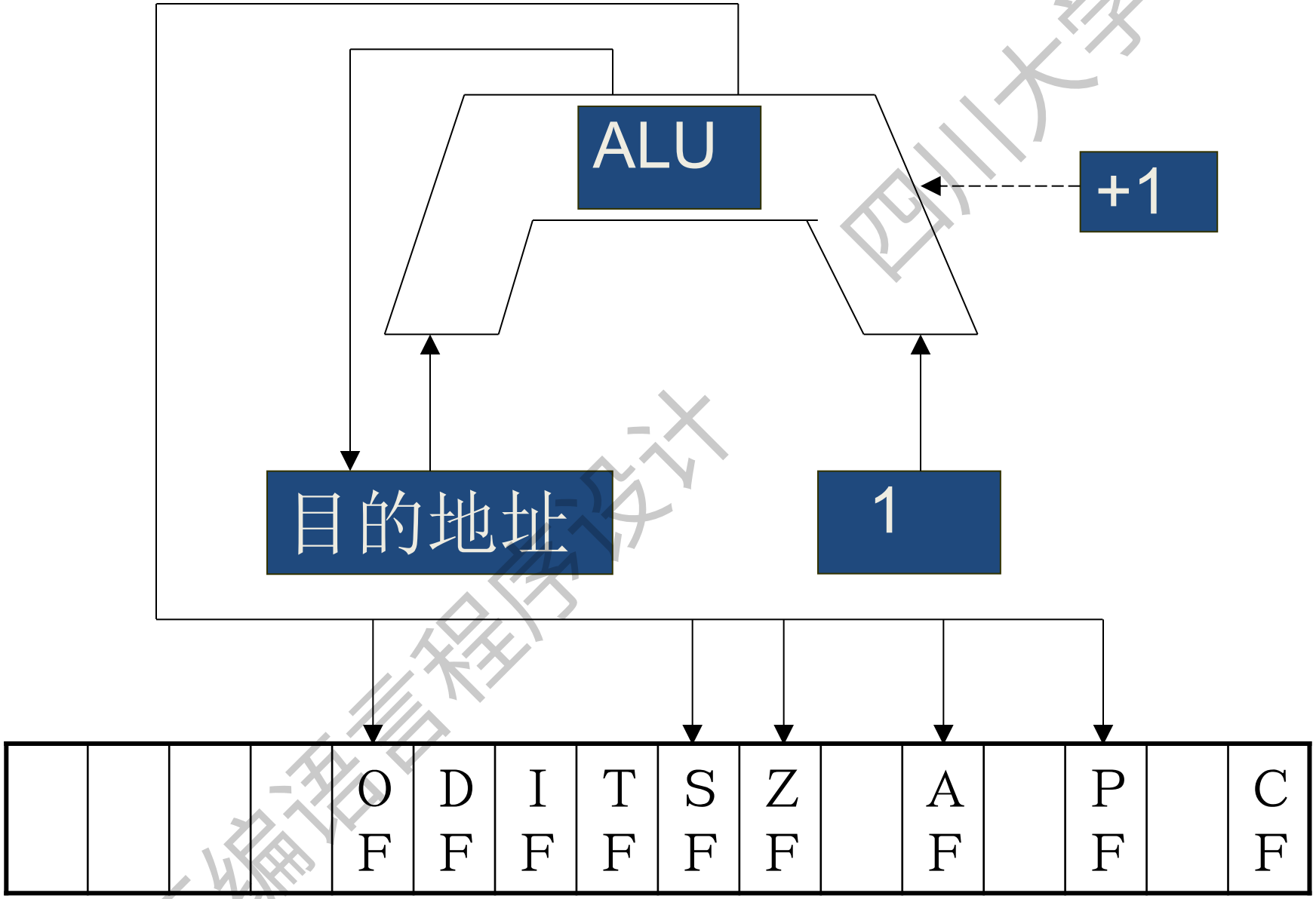
（3）加1指令

- 指令格式：INC　DEST
- 指令功能：DEST <=（DEST）+1
- 标志位影响：OF、SF、ZF、AF、PF
- 目的操作数地址可以为通用寄存器、内存单元；单位可以为字、字节。

（3）加1指令

- INC指令通常用于修改基址或变址寄存器，这样可以方便地访问数组中的元素，或者用于计数。

- INC指令的操作数通常理解为无符号数。



（3）加1指令

- INC指令与其他加法指令不同，它不影响CF标志，这是硬件机制决定的。如下例：

- MOV AL，0FFH
- INC AL

- （AL）=11111111B，
- CF=? **CF的取值在执行INC指令前后保持一致。**

（3）加1指令

- 为何设计为不影响CF？

```
        mov   bx, 0          ;完成80-bit操作数相加
        mov   cx, 5
        clc
lop1: mov   ax, tab1[bx]
        adc    ax, tab2[bx]   ; 生成 CF
        mov   res[bx], ax
        inc    bx             ; 不影响 CF
        inc    bx
        loop   lop1           ; 不影响 FR
```

（3）加1指令

- INC指令要影响OF标志，并且有一定规律可循。

- OF标志被置1时，表示刚执行完的指令使补码溢出，在INC指令中不外乎两种情况，正数加1后变成负数，负数加1后变成正数。

（3）加1指令

- INC　AL
- 正数变负数的情况：仅当（AL）= 01111111B，
- 加1后，（AL）= 10000000B，OF标志会被置1。

- 负数变正数的情况：仅当（AL）=11111111B，
- 加1后，（AL）= 00000000B

（3）加1指令

- 0不是正数，也不是负数，11111111B是－1的补码，-1+1=0是正确的（负+正，不会有溢出），所以没有溢出，OF被置0。

- 实际上OF被置1的情况只会出现在加1前（AL）= 01111111B时。

（3）加1指令

- 虽然INC要影响OF标志，但在INC指令中，操作数往往解释为无符号数，因而OF标志的实际意义不大。

- 如果考虑灵活使用，可以通过OF标志判断计数值是否超过半数，字节中为127，字中为32767。

（4）减法指令

- 指令格式：SUB　DEST，SRC
- 指令功能：DEST <=（DEST）-（SRC）
- 标志位影响：OF、SF、ZF、AF、PF、CF

（4）减法指令

- 源操作数可以为通用寄存器、内存单元、立即数；
- 目的操作数可以为通用寄存器、内存单元；
- 操作数可以字、字节为单位。

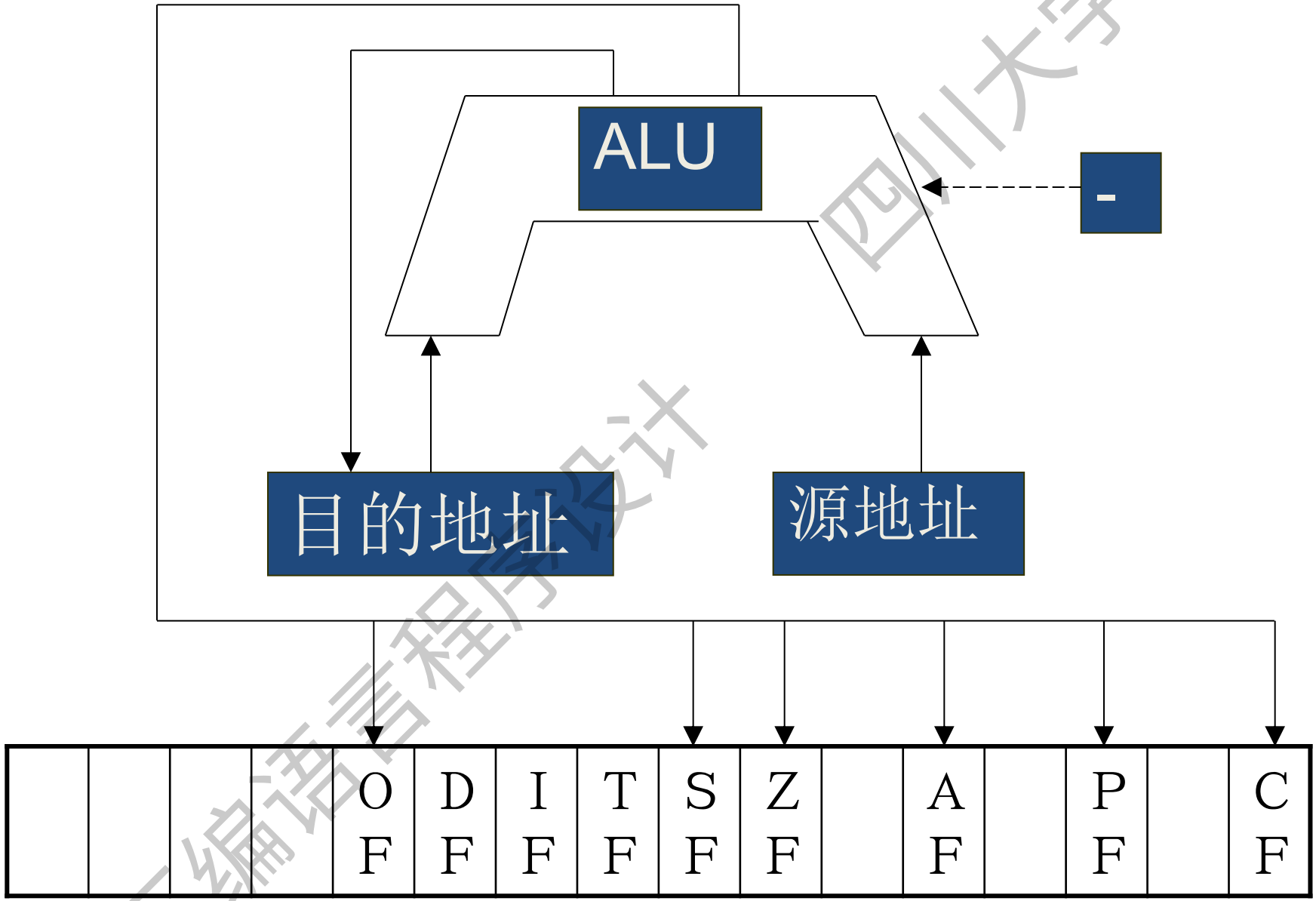
（4）减法指令

- 注意：
- 1）两个操作数最多只能有一个是内存单元。
- 2）对CF标志的判断必须使用无符号减法，不能使用补码相加的方法。

（4）减法指令

- 例
- 有两个字节单元A，B可以用以下指令实现
  （A）—（B）⟹ A

- MOV　AL，B
- SUB　A，AL



（4）减法指令

- 对于减法指令，标志位的有效性， 和加法指令一样，完全依赖于程序员对操作数的解释。

- 如果程序员把操作数解释为无符号数或者BCD码，那么CF、AF标志有效；

- 如果解释为带符号数，那么OF、SF标志有效；

（4）减法指令

- 如果解释为校验码，那么PF标志有效；
- ZF标志始终有效，用于判断运算结果是否为0。
- 课下请使用DEBUG观察减法指令的执行情况。

（5）带借位减法指令

- 指令格式：SBB　DEST，SRC

- 指令功能：
- DEST<=（DEST）-（SRC）-（CF）

- 标志位影响：OF、SF、ZF、AF、PF、CF

（5）带借位减法指令

- SBB指令对操作数的要求、对标志位的解释和SUB指令完全一致。

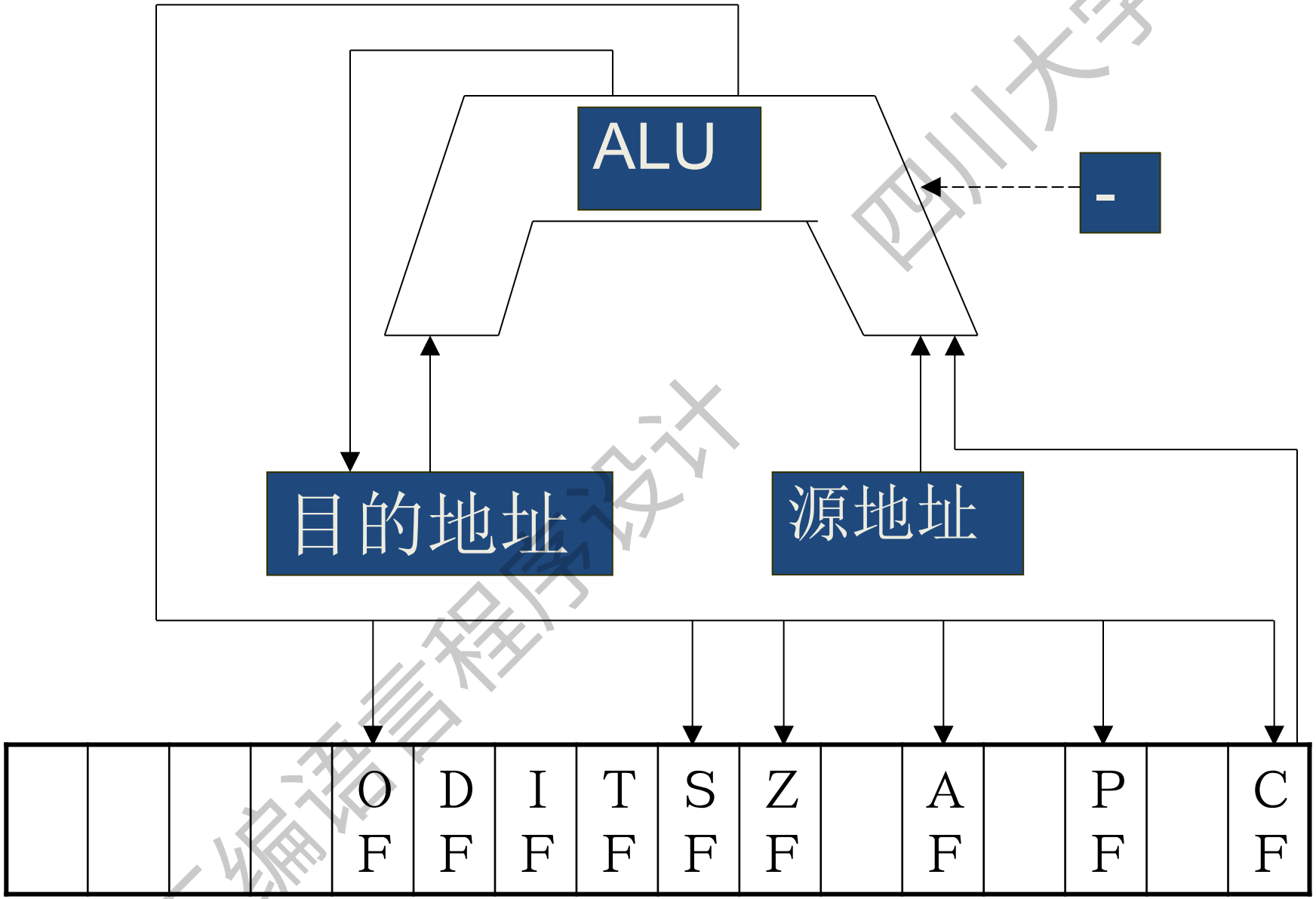
- SBB指令的主要用途和前面讲到的ADC指令相似。

（5）带借位减法指令

- ADC指令用于将低位单元加法产生的最高位进位引入到高位单元的加法中；SBB指令则是把低位单元减法产生最高位借位引入到高位单元的减法中。

- 和SUB指令结合使用完成多字节或多字数据的减法。

- 例子：多字节减法的例子留在课外自己构造。



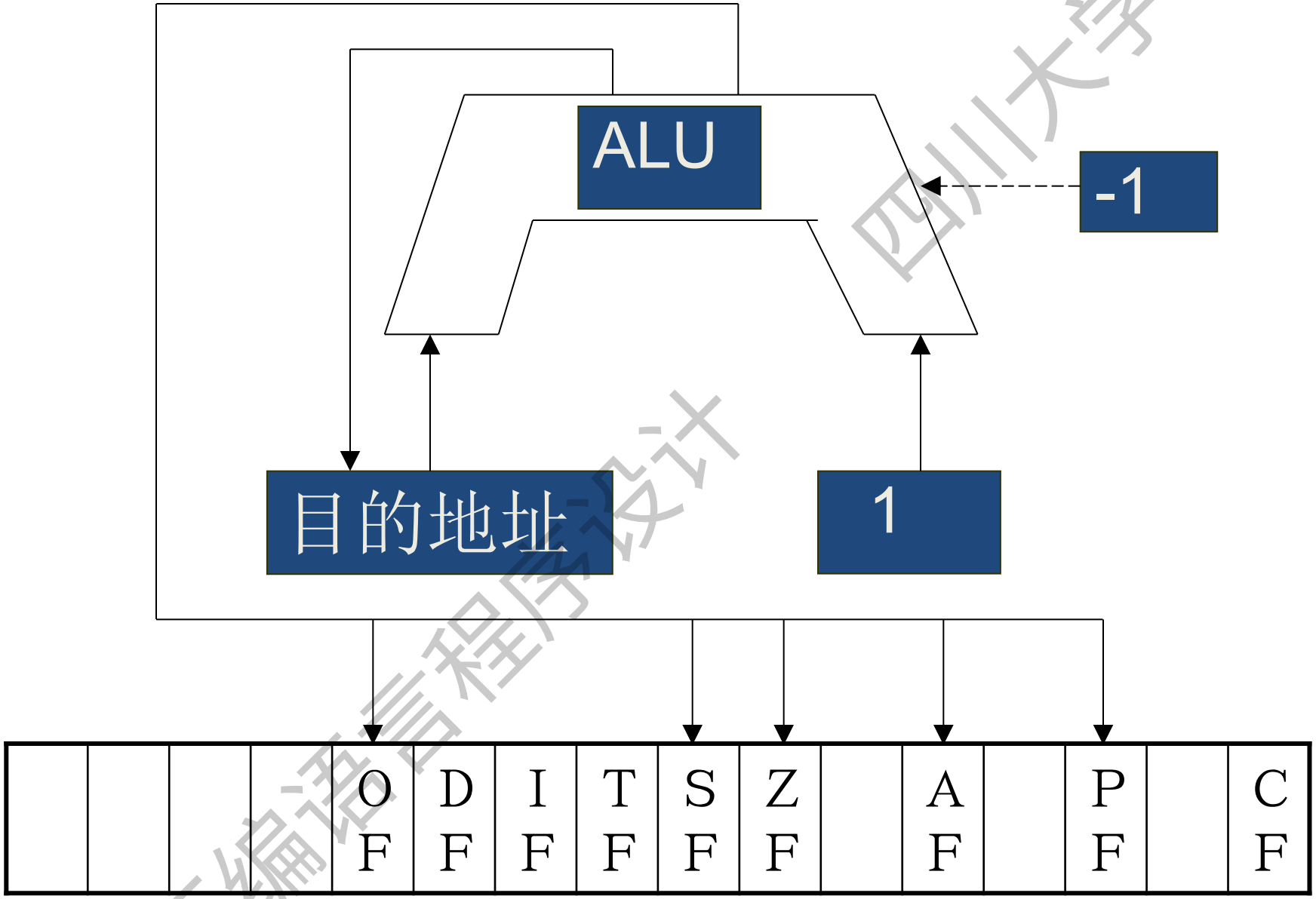
（6）减1指令

- 指令格式：DEC　DEST
- 指令格式：DEST <=（DEST）-1
- 标志位影响：OF、SF、ZF、AF、PF
- 目的操作数地址可以为通用寄存器、内存单元；可以字节、字为单位。

（6）减1指令

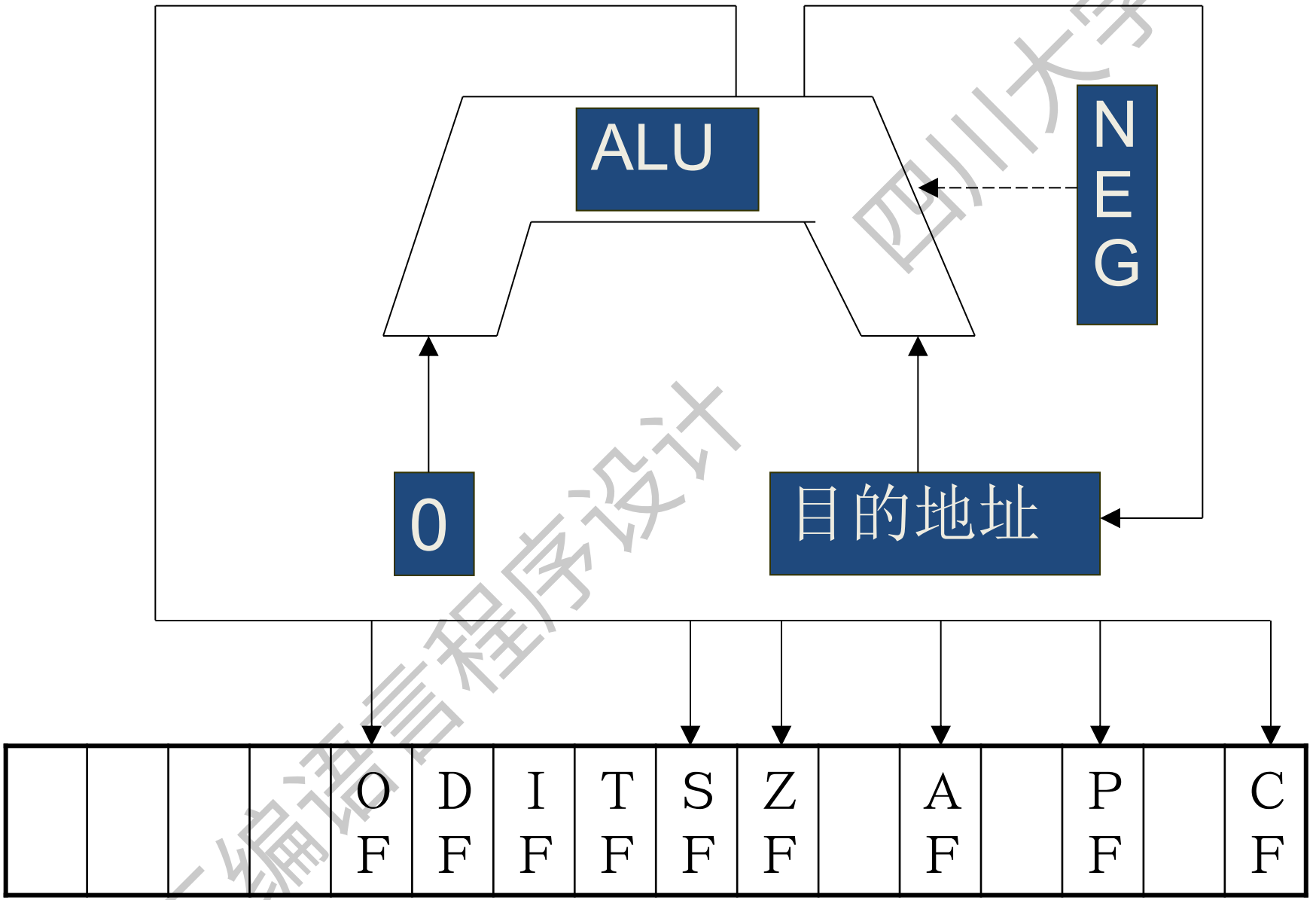
- 和INC指令的用途大致相同，用于计数或修改地址指针，只是方向和INC指令相反，操作数一般也解释为无符号数。

- 和INC指令一样，DEC指令不影响CF标志。



（7）求相反数指令

- 指令格式：NEG　DEST
- 指令功能：DEST <= -（DEST）
- 标志位影响：OF、SF、ZF、AF、PF、CF
- 目的操作数地址可以为通用寄存器、内存单元；可以字、字节为单位。
- 注意：操作数一般解释为补码



（7）求相反数指令

- NEG指令把操作数解释为带符号数的补码，完成的运算相当于用0减去操作数。

- 可能出现溢出的情况：-128、-32768等，是否溢出可由OF标志判断。

- 如果NEG指令的运算结果为0，那么CF标志置0，运算结果非0，CF标志置1。（CF的判断仍然按照无符号数的逻辑）

（7）求相反数指令

- 例 在DAW+2，DAW字单元存放有一个32位带符号数，DAW中存放的是低16位，求这个数的相反数并存入原单元中可用以下指令：

- NEG DAW
- MOV AX，0
- SBB AX，DAW+2
- MOV DAW+2，AX

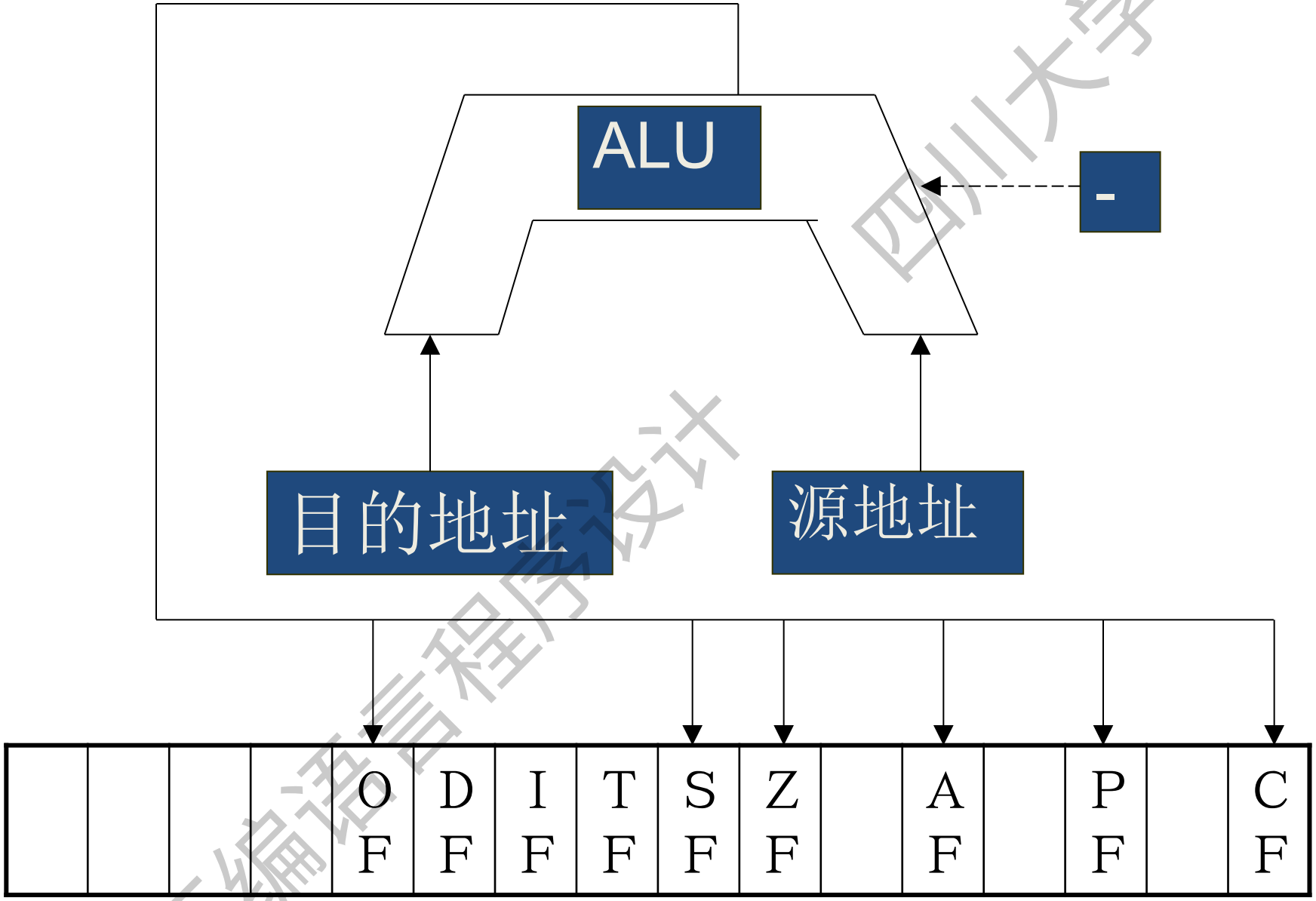
（8）比较指令

- 指令格式：CMP　DEST，SRC
- 指令功能：（DEST）-（SRC）
- 标志位影响：OF、SF、ZF、AF、PF、CF

（8）比较指令

- CMP指令的执行过程、对操作数的要求、对标志位的解释完全和SUB指令一致。

- 区别：CMP指令不保存运算结果。



（8）比较指令

- CMP指令仅使用减法操作来影响标志位，并不保存运算结果，因为它只关心标志位的状态，不关心实际的运算结果。

- 可以通过CF、OF、SF等标志位来判断两个无符号数或者两个有符号数的大小关系。

- 通常CMP指令和条件转移指令配合使用，在判断两个数的大小关系基础上实现程序的分支或者循环结构。

（8）比较指令

- 使用CMP指令对两个无符号数比较大小，判断标准为：

- CF=0：（DEST）-（SRC）最高位没有向更高位产生借位，（DEST）≥（SRC）

- CF=1：（DEST）-（SRC）最高位向更高位产生了借位，（DEST）<（SRC）

（8）比较指令

- 使用CMP指令对两个有符号数比较大小，判断标准如下：

- OF=SF：（DEST）≥（SRC）

- 1）OF=SF=0：减法运算没有溢出，符号位是正确的，正确的运算结果为正数或0。

- 2）OF=SF=1：减法运算有溢出，符号位是错误的，正确的运算结果为正数或0。

（8）比较指令

- OF≠SF：（DEST）<（SRC）

- 1）OF=0，SF=1：减法运算没有溢出，符号位是正确的，正确的运算结果为负数。

- 2）OF=1，SF=0：减法运算有溢出，符号位是错误的，正确的运算结果为负数。

（8）比较指令

- 如果在判断无符号数和有符号数大小的条件中加入对ZF标志的判断，会得到更丰富的判断结果。

- CMP指令仅仅通过减法运算影响标志位，它自己并不会判断这些标志位，标志位的判断是由条件转移指令来完成的。

- 这些相关的条件转移指令会在后面的课程中讲到。

第三章 8086/8088寻址方式及指令集(位操作类指令)

3.位操作类指令

- 包括逻辑运算指令、移位指令。
- 这些指令的共同特征：使用二进制位为基本处理单位。

（1）逻辑运算指令

- 完成二进制位的与、或、非、异或等逻辑运算。

1）逻辑与指令

- 指令格式：AND　DEST，SRC
- 指令功能：DEST <=（DEST）∧（SRC）
- 对操作数的要求和算术运算指令相同。

1）逻辑与指令

- 标志位影响：
- SF、ZF、PF解释与算术运算指令中相同，
- 无论运算结果为何值，CF、OF被强置为0，
- AF不确定，它会受到影响，但是指令对AF不作任何解释。

“不确定”标志位

- 如果某标志位被某条指令定义为“不确定”，则该标志位可能会因为执行该指令而变化，但是该标志位并没有任何实际意义，不能使用。

1）逻辑与指令

- 例1.按位相与
- （AL）=00101101B
- （AH）=10010111B

- AND　AL，AH

- 00101101
- ∧　10010111
- 00000101

1）逻辑与指令

- 指令执行后，（AH）不变，（AL）=00000101B

- SF=0，ZF=0，PF=1
- CF=OF=0（强行置0）
- AF不确定，无意义

2）逻辑或指令

- 指令格式：OR　DEST，SRC
- 指令功能：DEST <=（DEST）∨（SRC）
- 标志位影响：SF、ZF、PF解释和逻辑与指令相同，CF、OF置0，AF不确定
- 对操作数的要求和算术运算指令相同。

2）逻辑或指令

- 例2.按位相或
- （BL）=11100010B
- （BH）=00111011B

- OR　BL，BH

- 11100010
- ∨　00111011
- 11111011

2）逻辑或指令

- 指令执行后，（BH）不变，（BL）=11111011B

- SF=1，ZF=0，PF=0
- CF=OF=0（强行置0）
- AF不确定，且无意义

3）逻辑异或指令

- 指令格式：XOR　DEST，SRC
- 指令功能：DEST <=（DEST）⊕（SRC）
- 标志位影响：SF、ZF、PF解释同前面的逻辑运算指令，CF、OF置0，AF不确定。
- 对操作数的要求和算术运算指令相同。

3）逻辑异或指令

- 例3.按位异或
- （DL）=11010011B
- （DH）=10100010B

- XOR　DH，DL

- 11010011
- ⊕ 10100010
- 01110001

3）逻辑异或指令

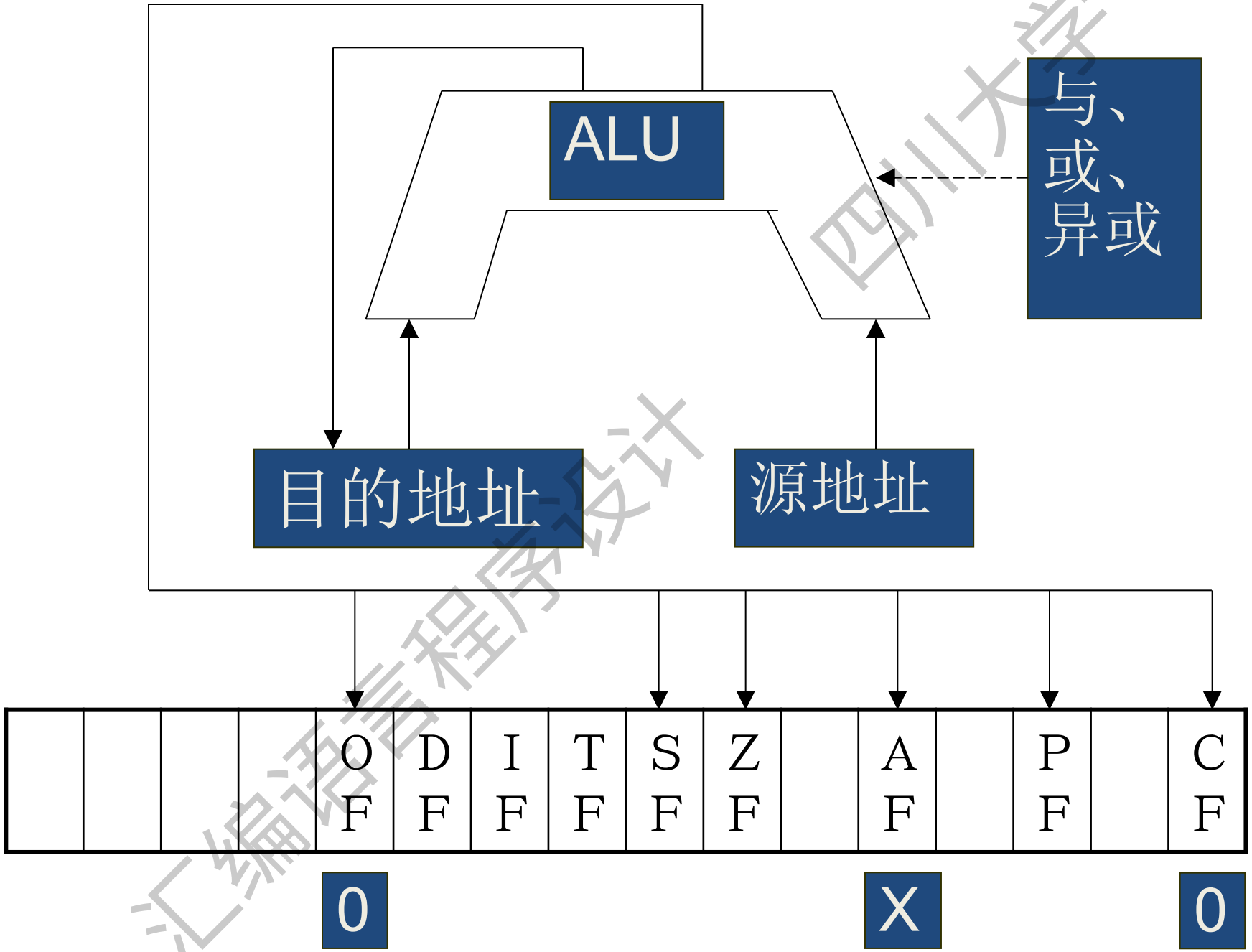
- 执行指令后，（DH）不变，（DL）=01110001B

- SF=0，ZF=0，PF=1
- CF=OF=0（强行置0）
- AF取值不定，且没有意义

双操作数逻辑运算指令

- 前面3条逻辑运算指令都是双操作数指令，除了具体的运算过程不同外，对操作数的要求，对标志位的影响、解释都是相同的。

- 这3条逻辑运算指令的执行过程也很相似，这里把它们的运算过程统一在同一个说明图中表示：



4）逻辑非指令

- 指令格式：NOT　DEST
- 指令功能：DEST <= ㄱ（DEST）
- 标志位影响：无
- 对操作数的要求和单操作数的算术运算指令一样。

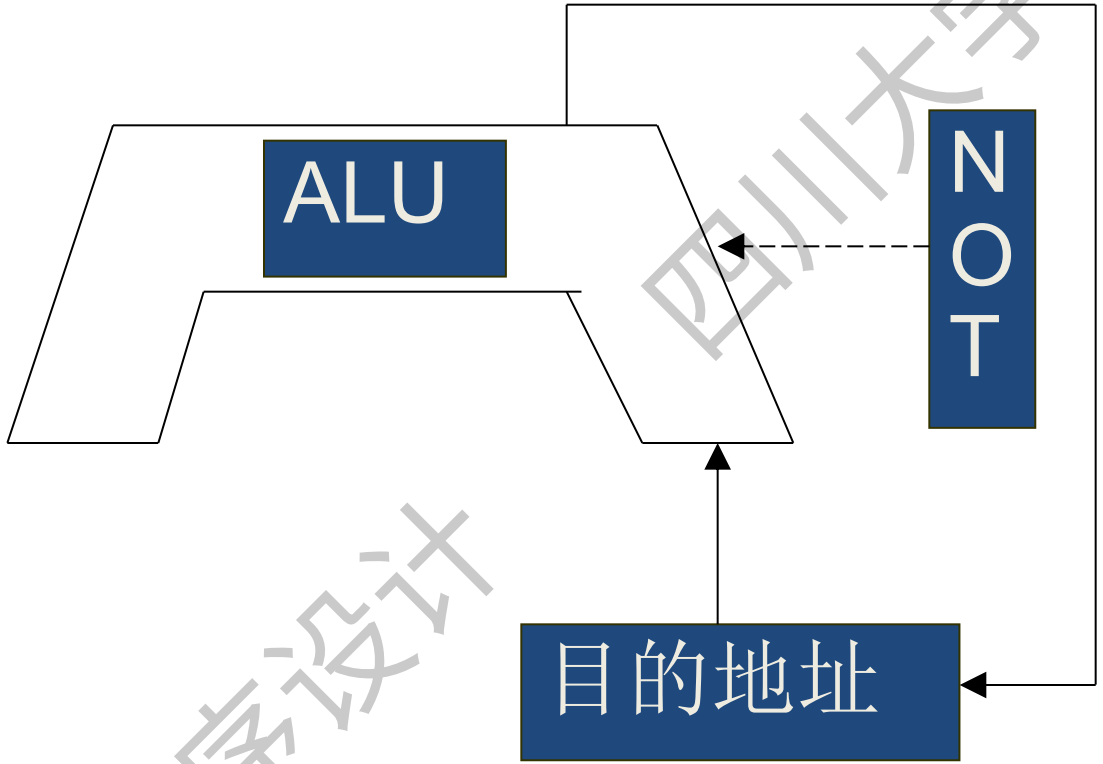
4）逻辑非指令

- 例4.按位取反
- （AL）=01011100B
- NOT　AL

- ┐　　01011100
- 　　10100011

- 执行指令后，（AL）=10100011B

- 没有任何标志位受影响。



				OF	DF	IF	TF	SF	ZF		AF		PF		CF

逻辑运算指令经典分析错误

- 易错题目：

- STC
- OR　AL，1
- ADC AL，BL

- 执行ADC时CF标志位为多少？

逻辑运算指令的应用

- 除了完成通常的逻辑运算外，逻辑运算指令还常用于针对存储单元中数据位的操作。

- 对于这方面的应用，我们看几个例子就能深刻理解“位操作指令”中“位操作”的含义。

取位操作

- 例1.取位操作
- （AL）=10100011B
- AND　AL，0FH

- 执行指令后，（AL）=00000011B

- 以立即数00001111为“取位模板”，通过与运算把（AL）的高4位清0，保留低4位，也就是把（AL）的低4位从（AL）中分离出来。

置位操作

- 例2.置位操作
- （AL）=00001000B
- OR　AL，11H

- 执行指令后，（AL）=00011001B

- 以立即数00010001为“置位模板”，通过或运算把（AL）的第0位和第4位置为1，同时不影响（AL）中的其他数据位。

位变反操作

- 例3.位变反操作
- （AL）=01110101B
- XOR　AL，00100010B

- 执行指令后，（AL）=01010111B

- 使用立即数00100010B为“变反模板”，通过异或运算把（AL）的第1位、第5位变反，同时不影响其他数据位。

逻辑运算指令

- 例 可用以下程序段实现将AL的高4位与BL的低4位进行组合：

- AND AL，0F0H
- AND BL，0FH
- OR AL，BL

逻辑运算指令

- 例 可用以下程序段实现将标志寄存器的第8位TF为置“1”：
- PUSHF
- POP AX
- OR AX，100H
- PUSH AX
- POPF

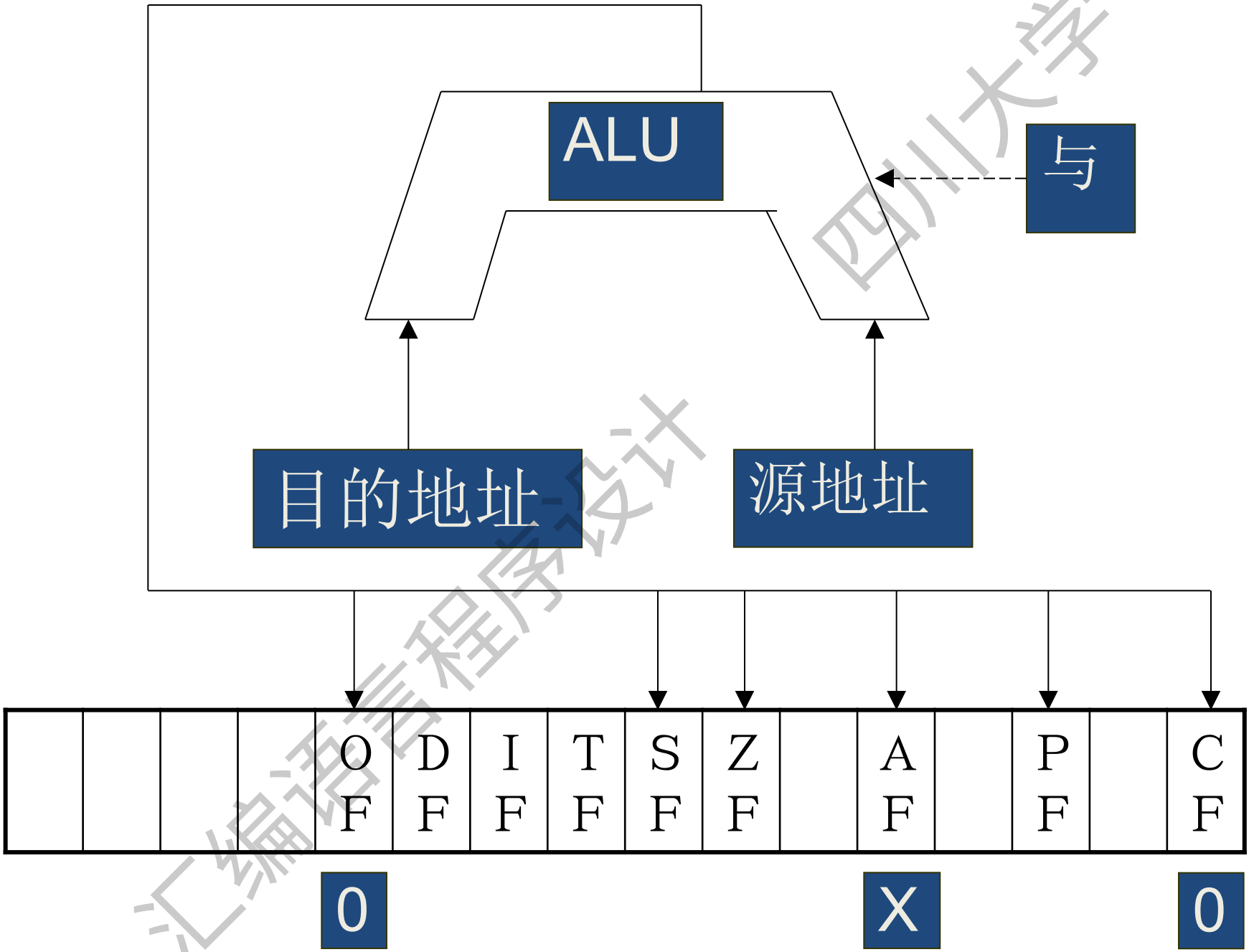
（2）测试指令

- 指令格式：TEST　DEST，SRC
- 指令功能：（DEST）∧（SRC）
- 标志位影响：SF、ZF、PF解释与AND指令一致，CF、OF强行置为0，AF不确定。

（2）测试指令

- TEST指令在各方面几乎和AND指令完全一样，唯一的区别是TEST指令不保存运算结果。

- 其对应关系与CMP指令和SUB指令的对应关系非常相似。



（2）测试指令

- 测试指令通常用于对存储单元中某些感兴趣的数据位进行测试，所用的运算就是与运算。

- 测试指令对运算结果不感兴趣，通常只对标志位，特别是ZF标志感兴趣。

（2）测试指令

- 例4.测试（AL）中的第2位是否为1
- TEST　AL，00000100B
- 取位模板中除第2位为1外，其余数据位全部为0，那么与运算的结果除了第2位可能不为0外，其余位一定为0。
- 运算结果的第2位是否为0由（AL）中的第2位决定。

（2）测试指令

- 运算结果的第2位是否为0等价于整个运算结果是否为0。

- 可以通过判断ZF标志是否为1来判别（AL）的第二位是否为0。

（2）测试指令

- 适当设置“取位模板”，则可对指定存储单元中感兴趣的数据位进行测试，并可通过ZF标志来判断该数据位的取值。

- 程序设计中，TEST指令通常和条件转移指令结合使用。通过特定存储单元中特定数据位的不同取值来实现程序中的分支或循环结构。

（2）测试指令

- CMP指令和TEST指令都主要用于实现程序的分支和循环结构，只是方法不同，CMP指令通过数值比较来影响标志位，TEST指令通过数据位测试来影响标志位。

位操作示例

- 假设第35H号端口（8-bit）的每个bit外接一个LED，一个bit为1则点亮对应的LED，为0则熄灭对应的LED，现要求点亮第2个LED，但同时不影响其它LED的状态。

- IN　　AL, 35H
- OR　 AL, 02H
- OUT　35H, AL

位操作示例

- 假设第34H号端口（8-bit）中的各bit反映键盘上不同的状态，第1位反映CAPS-LOCK的状态。现要求依据当前CAPS-LOCK的状态实现分支程序结构。

- IN　　AL, 34H
- TEST　AL, 01H
- JZ　　L1
- JMP　L2

（3）移位/循环移位指令

- 移位指令的功能是对指定存储单元中的二进制编码按照指定方式、以数据位为移动单位向左或向右移动。
- 共通的指令格式：OPR　DEST，COUNT
- OPR：操作助记符，指明移位的方式。
- DEST：目的操作数地址，必须为通用寄存器或存储单元，可以为字节、字。
- COUNT：移位次数，为1时可使用常数1表示，如果大于1，必须使用寄存器CL中的内容给出。

1）算术移位指令

- 算术左移指令格式：SAL　DEST，COUNT
- 指令功能：将（DEST）左移COUNT位，移出的最低位保存到CF标志，空出的低位则补充0。
- 标志位影响：OF、SF、ZF、PF、CF 、AF
- **操作数解释为补码**

1）算术移位指令

- 除CF的解释有变化外（本质未变），其余标志的含义与算术运算指令中完全一致。AF标志不确定。

- 特别注意OF标志，仅当COUNT=1时，它才有意义。

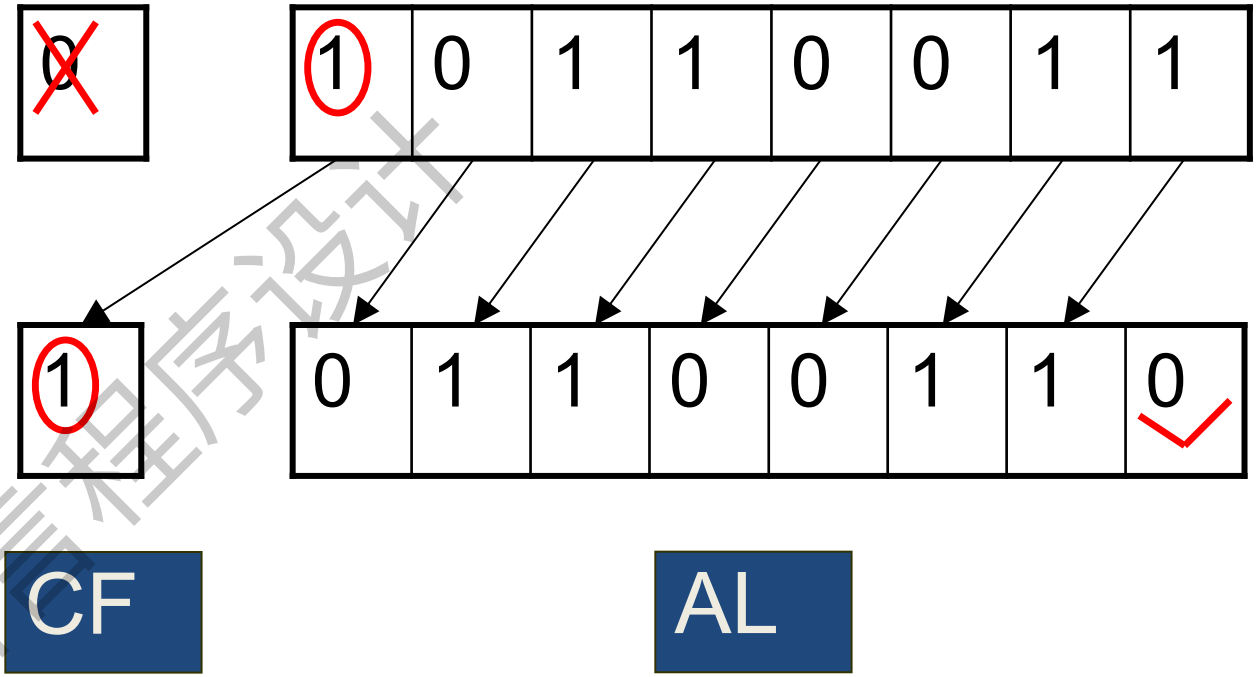
- 既然操作数解释为补码，为何仍需要CF标志？ 长补码移位操作

- 例1.将（AL）算术左移一位。

- SAL　AL，1

- 移位前：

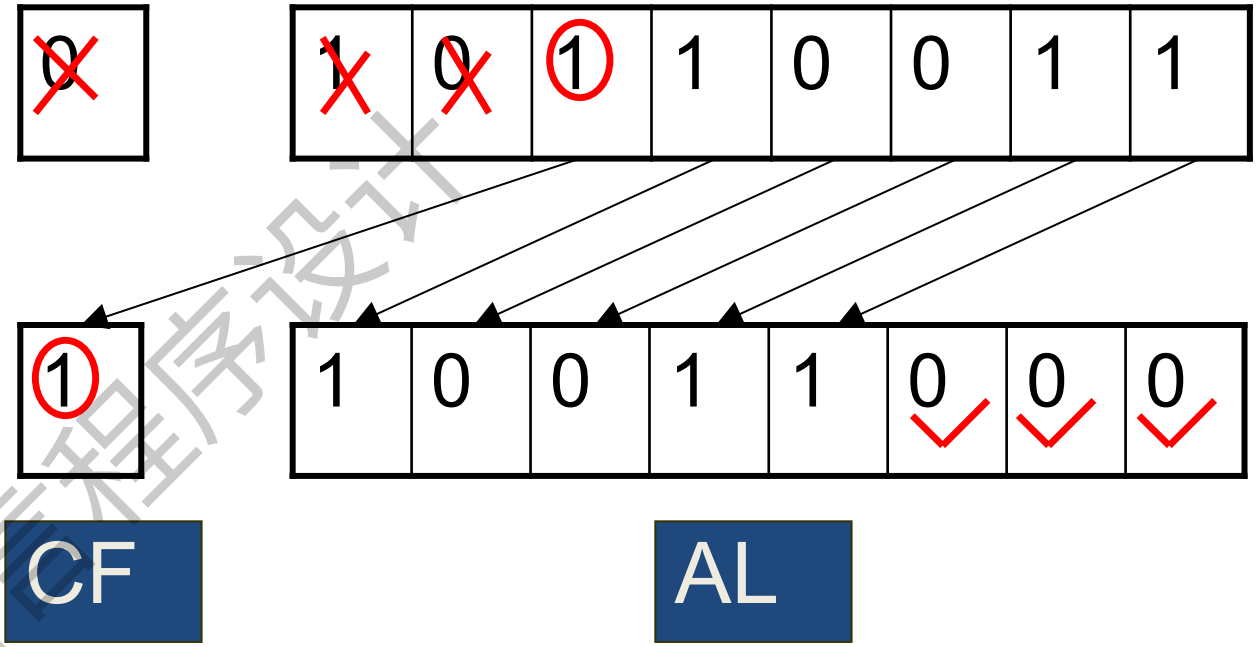
- 移位后：



- OF = 1

例2.将（AL）算术左移3位。

- MOV　CL，3
- SAL　AL，CL
- 移位前：



- 移位后：
- OF标志：无意义，因为硬件上移动3位同时完成，无法觉察符号位的多次变化。

算术右移指令

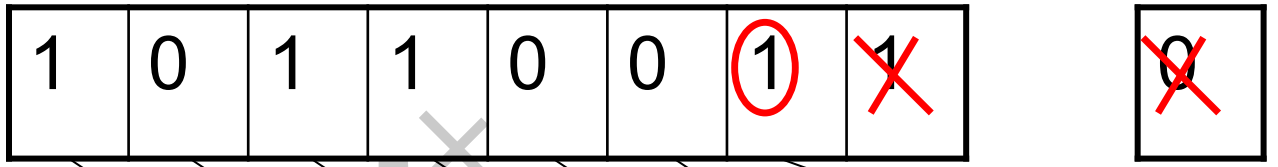
- 指令格式：SAR　DEST，COUNT
- 指令功能：将（DEST）右移COUNT位，移出的最高位保存到CF标志，空出的高位则补充原来的符号位。
- 标志位影响：OF、SF、ZF、PF、CF 、AF
- 操作数解释为补码

算术右移指令

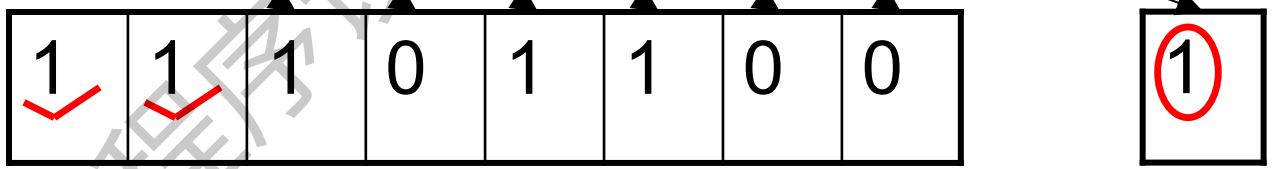
- 除CF的解释有变化外，其余标志的含义与算术运算指令中完全一致。
- AF标志不确定。

例3.将（AL）算术右移2位。

- MOV　CL，2
- SAR　AL，CL
- 移位前：



- 移位后：



AL　　CF

算术移位

- 算术移位指令是针对带符号数移位操作的，每左移一位相当于把补码乘以2，每右移一位相当于把补码除以2。

算术右移指令

- 右移时在数据左侧补充符号位是为了保持补码特性，不改变符号。

- 当右移操作进行了多次，符号位占据了整个操作数空间以后，数值不会发生变化。

算术右移指令

- 正数右移所得最小值为0，负数右移所得最小值为-1。

- 算术右移不会发生溢出的现象，始终都有OF=0。

算术左移的溢出

- 算术左移可能会出现溢出的现象。

- 例如将补码10110011算术左移一位。
- 结果为：01100110
- OF=1，发生了溢出。

算术移位实现补码乘除法

- 例3．2．18 AX中已存放一个带符号数，若要完成（AX）*3/2运算，可用以下程序段实现：

- MOV DX，AX
- SAL AX，1 ；（AX）乘2
- ADD AX，DX ；（AX）乘3
- SAR AX，1 ；完成（AX）*3/2

2）逻辑移位指令

- 逻辑左移指令：SHL　DEST，COUNT

- 指令功能：将目的操作数地址中的数据左移COUNT位，移出的最低位保存到CF标志，空出的低位则补充0。

- COUNT为1时可以用常数表示，如果大于1，必须用CL寄存器中的内容表示。

- **操作数解释为无符号数。**

SHL指令与SAL指令

- 无符号数
- 逻辑左移指令实际上和算术左移指令是同一条指令，机器指令代码完全一致。

- 在DEBUG中输入指令时只有SHL指令合法，而SAL指令非法。

- 但是在源程序中SAL指令是可以使用的。

逻辑右移指令

- 指令格式：SHR　DEST，COUNT

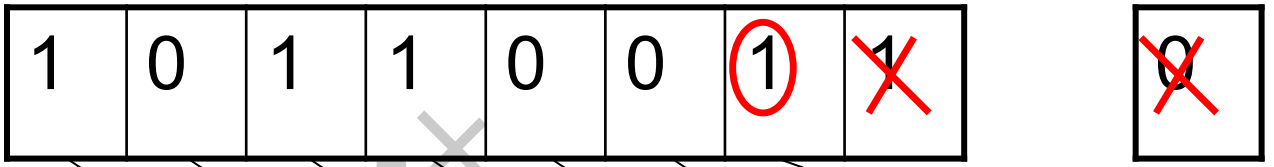
- 指令功能：将目的操作数地址中的数据右移COUNT位，移出的最高位保存到CF标志，空出的高位则**补充0**（与算术移位指令不同）。

- COUNT为1时可以用常数表示，如果大于1，必须用CL寄存器中的内容表示。

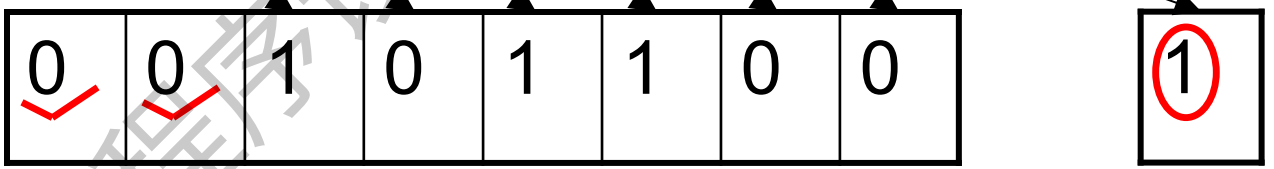
- **操作数解释为无符号数。**

例4.将（AL）逻辑右移2位。

- MOV　CL，2
- SHR　AL，CL
- 移位前：



- 移位后：



AL　　CF

逻辑移位指令

- 逻辑移位指令是针对无符号数移位操作的，每左移一位相当于把无符号数乘以2，每右移一位相当于把无符号数除以2。

- 因为无符号数移位操作不需要考虑符号，所以逻辑右移操作是在最高位（最左）补充0（算术右移操作是在最高位补充符号位）。

逻辑移位指令

- 虽然逻辑移位指令把数据解释为无符号数，但仍影响OF标志，且仍按照带符号数溢出的判断逻辑进行设置。
- 因此，SHL、SHR指令都可能出现带符号数溢出的情况。
- 但是，程序员往往不关心SHL，SHR中的OF标志。

3）循环移位指令

- 循环左移指令格式：ROL　DEST，COUNT
- 指令功能：把目的地址中的数据循环左移COUNT位，从高位（左）移出的数据位都补充到低位（右），移出的最低位保存到CF标志位。
- 标志位影响：CF、OF
- CF标志用于保存移出的最低位。

3）循环移位指令

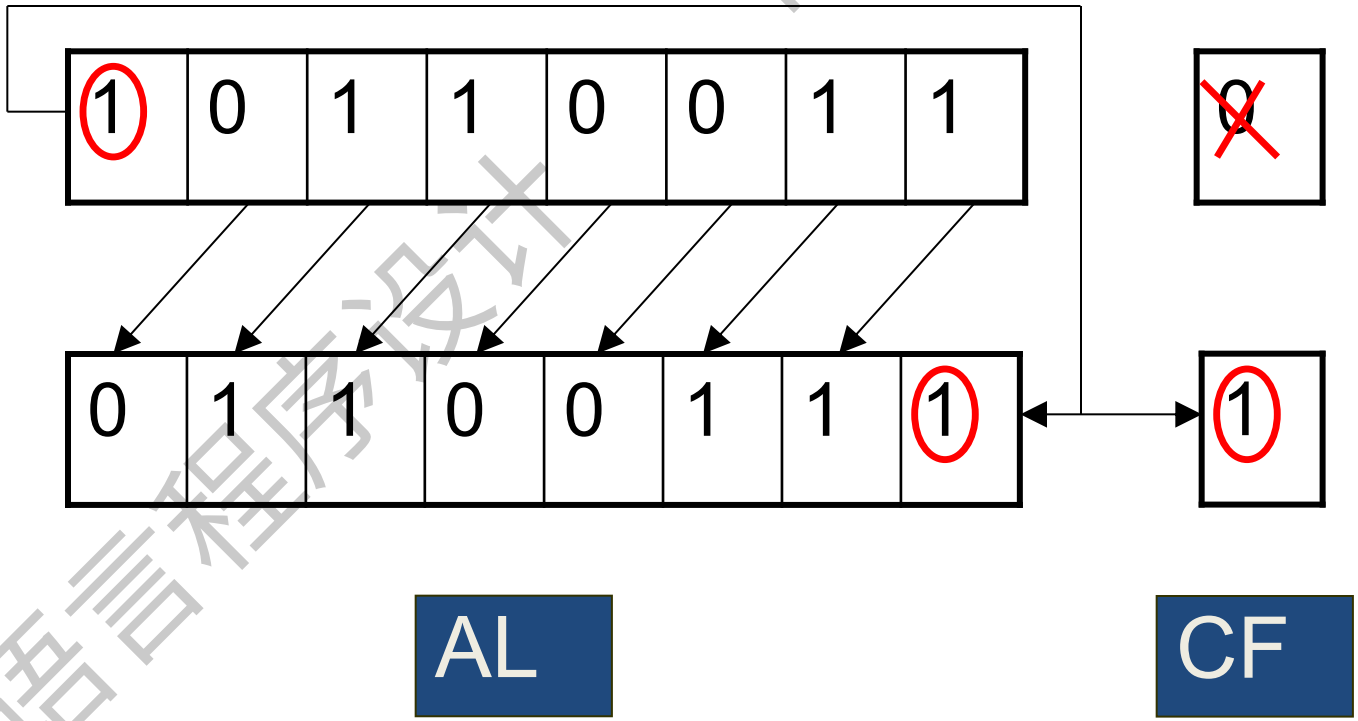
- 如果COUNT=1，OF标志有意义，如果移位前后数据的符号位发生了变化，OF=1；如果符号位没有发生变化，OF=0。

- OF在这里解释为补码溢出标志合适吗？

- 如果COUNT>1，OF标志不确定（没有意义）。

例5.将（AL）循环左移1位。

- ROL　AL，1
- 移位前：
- 移位后：
- OF=1



循环右移指令

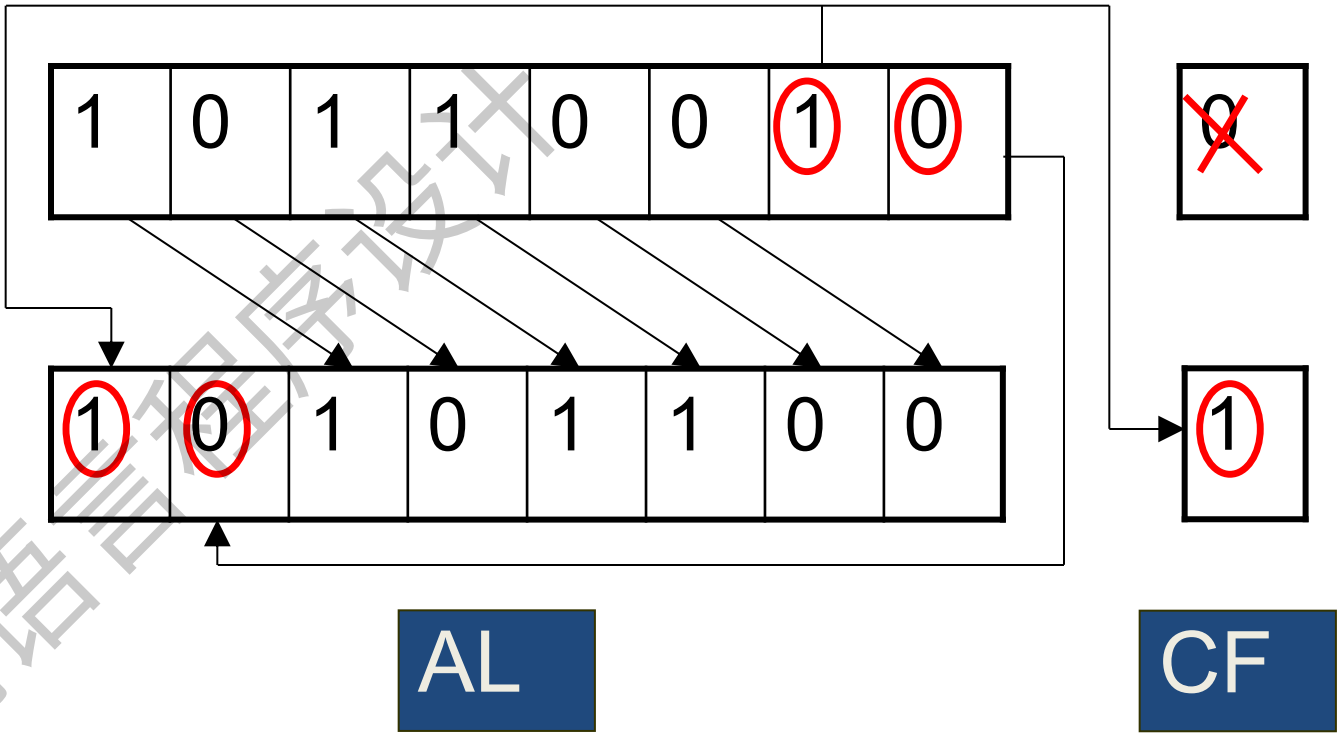
- 循环右移指令格式：ROR　DEST，COUNT

- 指令功能：把目的地址中的数据循环右移COUNT位，从低位（右）移出的数据位都补充到高位（左），移出的最高位保存到CF标志位。

- 标志位影响：CF、OF，标志位的解释和循环左移指令中完全一致。

例6.将（AL）循环右移2位。

- MOV　CL，2
- ROR　AL，CL
- 移位前：
- 移位后：
- OF不确定



ROL 或 ROR 的应用

- 假设 (AL)为来自于端口的待测试数据。
- 假设 (CL) 中指定了待测试数据中的待测试位。

```
TESTSTATUS  PROC
            PUSH  BX
            MOV   BL, 01H
            ROL   BL, CL
            TEST  AL, BL
            POP   BX
            RET
TESTSTATUS  ENDP
```

带进位循环左移指令

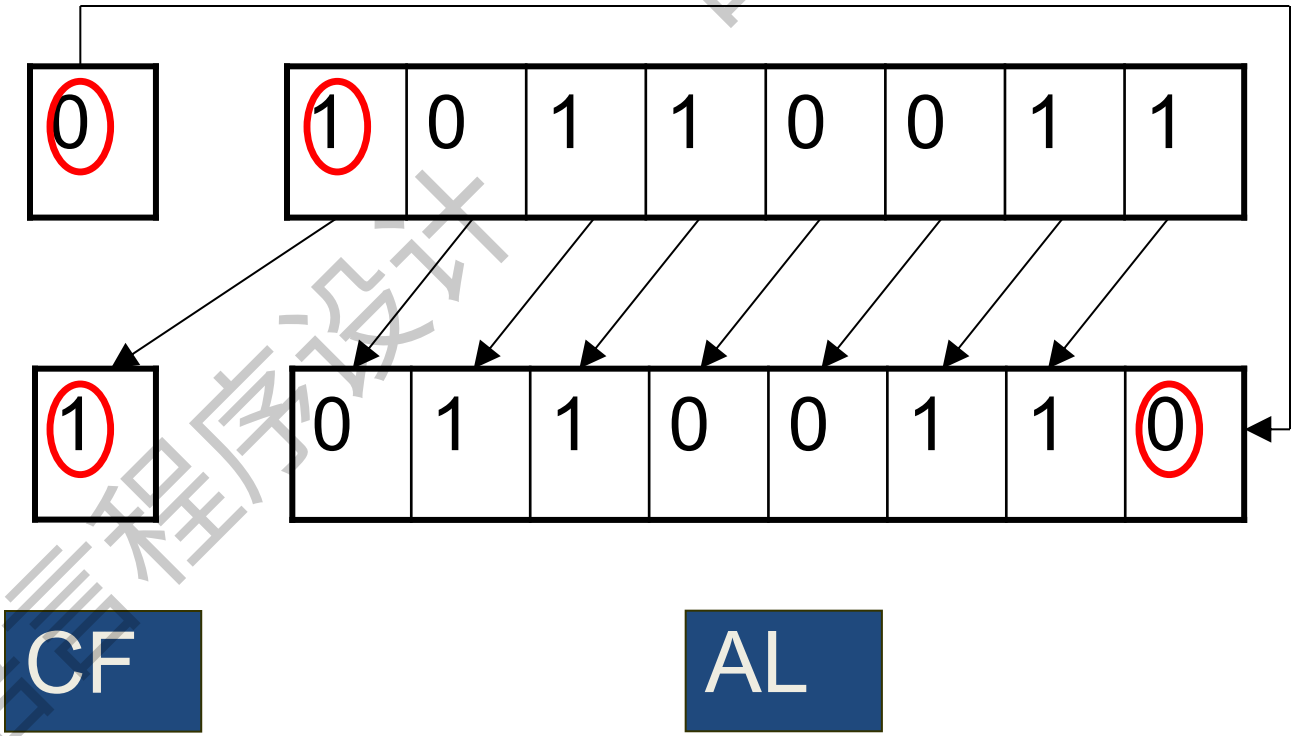
- 指令格式：RCL　DEST，COUNT

- 指令功能：将CF与目的操作数看作一个整体，CF作为最高位（最左），对整体数据循环左移COUNT位。

- 标志位影响：CF标志作为数据最高位参加移位操作；OF标志位的解释与循环移位指令一致。

例7.将（AL）带进位循环左移1位。

- RCL　AL，1
- 移位前：
- 移位后：
- OF=1



带进位循环右移指令

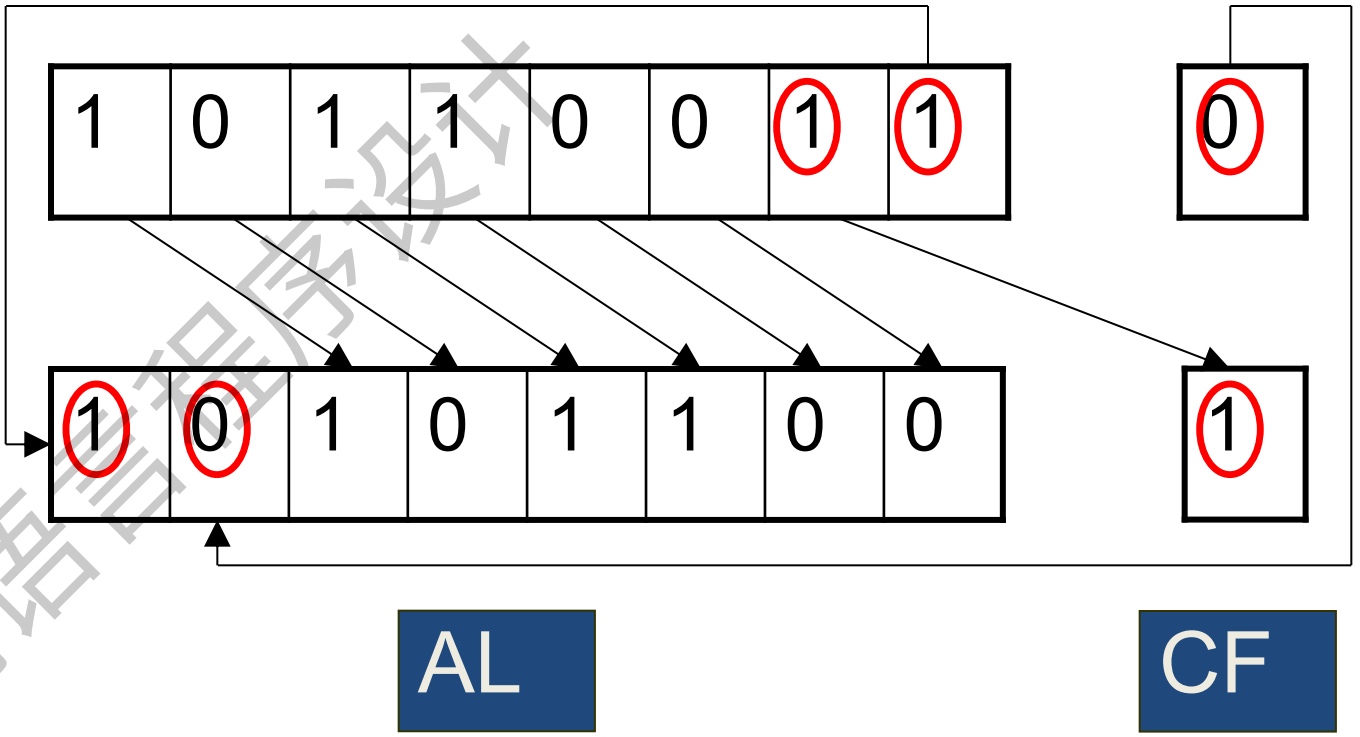
- 指令格式：RCR　DEST，COUNT

- 指令功能：将CF与目的操作数看作一个整体，CF作为最低位（最右），对整体数据循环右移COUNT位。

- 标志位影响：CF标志作为数据最低位参加移位操作；OF标志位的解释与循环移位指令一致。

例8.将（AL）带进位循环右移2位。

- MOV　CL，2
- RCR　AL，CL
- 移位前：
- 移位后：
- OF不确定



多字、多字节无符号数移位

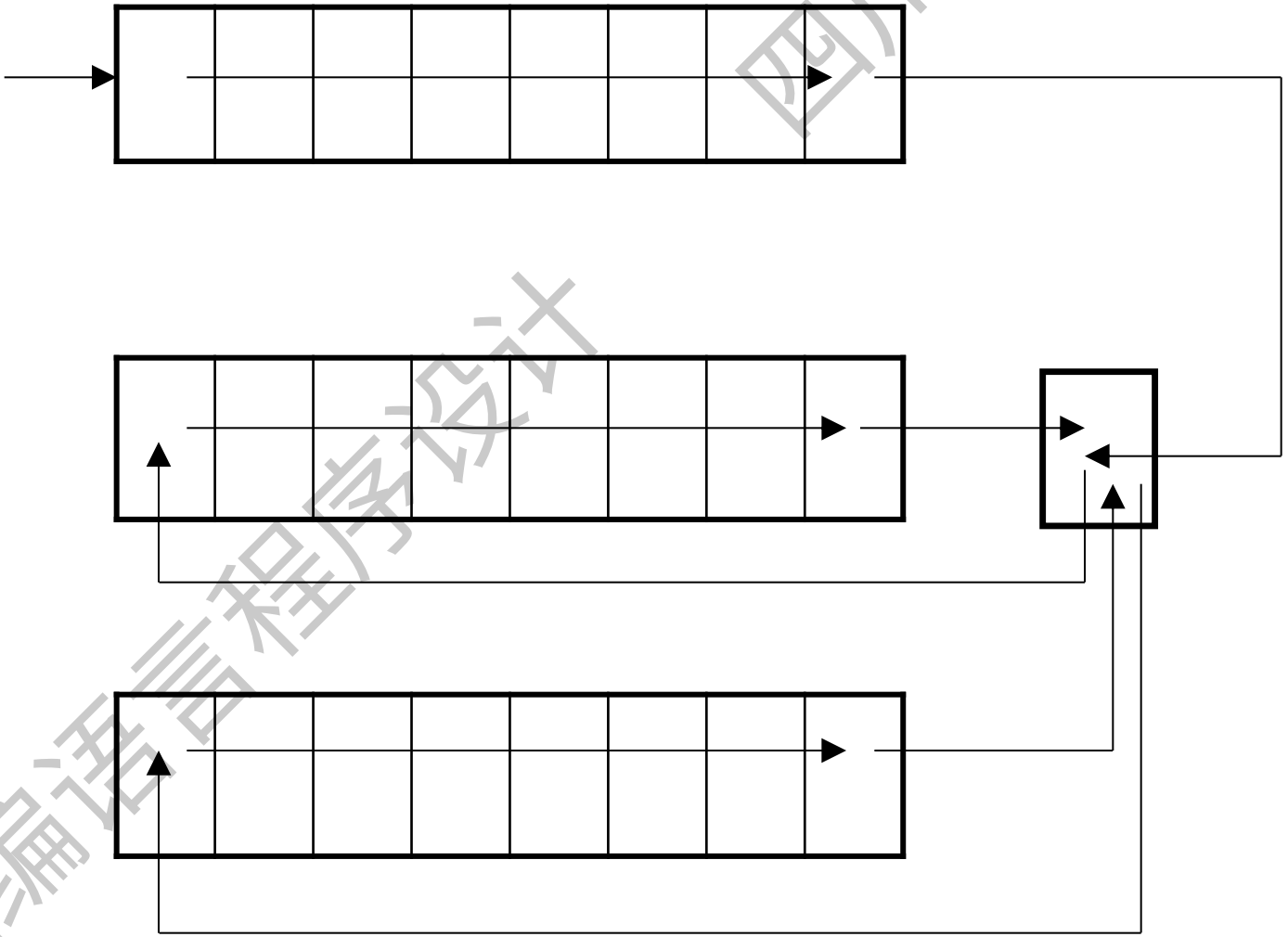
- 逻辑移位与带进位移位指令结合使用可完成对**多字或多字节组成的无符号数**进行移位操作。
- 右移操作无溢出。
- 左移操作可能会溢出，应在**最后一次移位**（最高字或字节的左移）后判断溢出状态，应判断CF标志。

多字、多字节带符号数移位

- 算术移位与带进位移位指令结合使用可完成对**多字或多字节组成的带符号数**进行移位操作。
- 右移操作无溢出。
- 左移操作可能会溢出，应在**最后一次移位**（最高字或字节的左移）后判断溢出状态，应判断OF标志。

多字节或多字数据算术或逻辑右移：

- 高字节：
- 低字节：



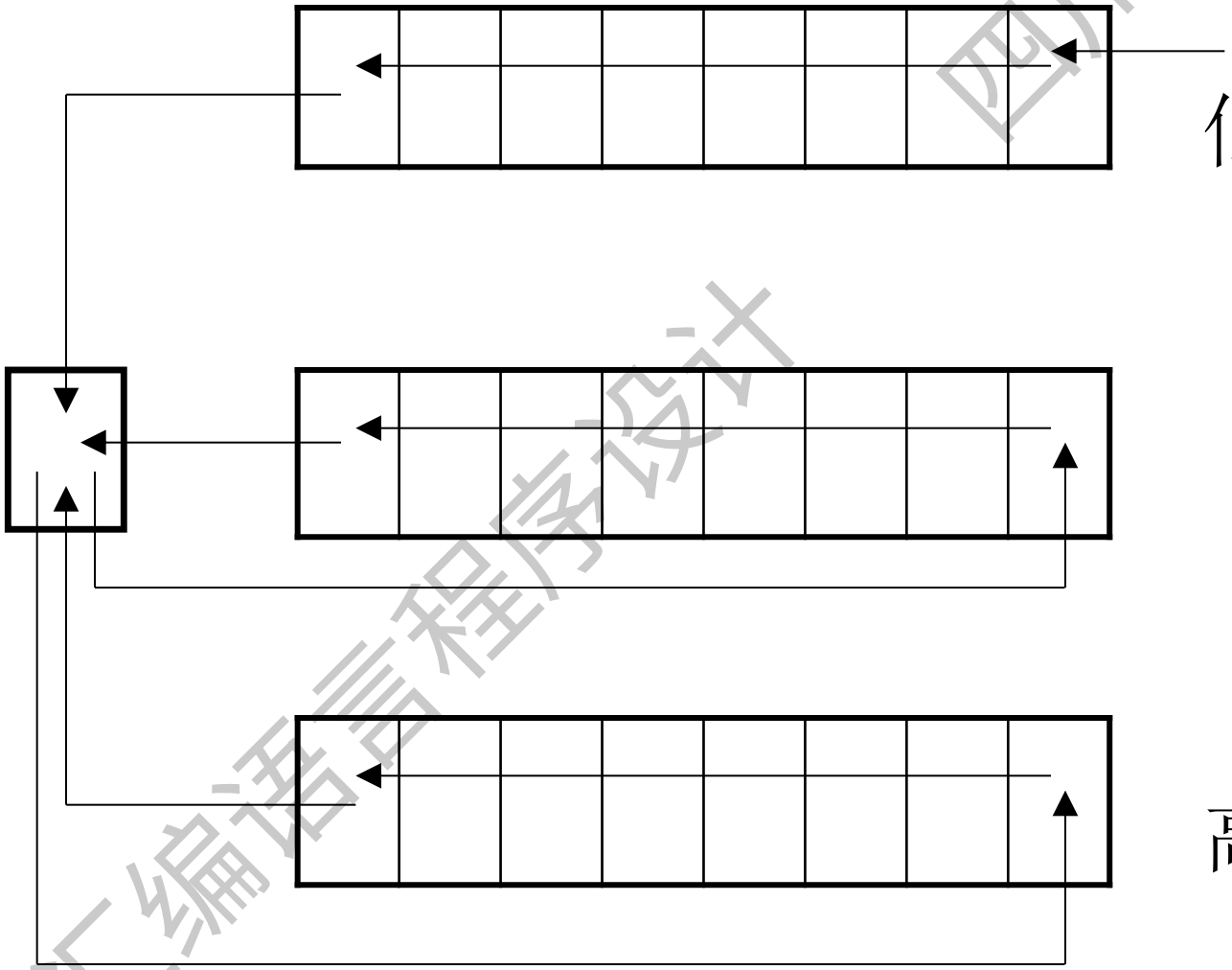
多字、多字节无符号数移位

- 例 由三个字构成的一个无符号数从高位到低位依次存放在M+4，M+2，M字单元中，若要将该数右移一位，则可用以下程序段：

- SHR M+4，1
- RCR M+2，1
- RCR M，1

多字节或多字数据算术或逻辑左移：

- 低字节
- 高字节



多字、多字节移位

- 注意，对于多字或多字节的移位，只能每次移动一位，COUNT=1，一次一次的完成。

- 对**左移操作**而言在**最后一次移位**时**判断是否溢出**。

多字、多字节移位

- 每次只移一位的原因：
- 1）CF每次只能容纳一个被移出的数据位。
- 2）对带符号数而言：同时移动多位，OF将无意义。

第三章 8086/8088寻址方式及指令集(处理器控制指令)

4. 处理器控制指令

- 用于改变处理器运转模式或改变处理器对特定事件的处理方式的指令。

1）标志位操作指令

- 这类指令都是无操作数指令，使用的都是隐含操作数，把特定的标志位置0或置1。

（I）清除进位标志指令（Clear carry flag）

- 指令格式：CLC
- 指令功能：将CF标志清0。

清除进位标志指令

- 例1.完成三字数据的加法
- CLC
- ADC　AX，BX
- ADC　DX，CX
- ADC　 SI，　DI

- 功能：（SI，DX，AX）+（DI，CX，BX）

- 如果使用程序的循环结构，那么只需要一条ADC指令。

（II）进位标志置位指令（Set carry flag）

- 指令格式：STC
- 指令功能：将CF标志置1

（III）进位标志取反指令（Complement carry flag）

- 指令格式：CMC
- 指令功能：CF标志位取反。

进位标志取反指令

- 例2.将（AL）逐位取反，取反结果存放到AH寄存器

- SHL　AL，1
- CMC
- RCL　AH，1
- 循环8次把所有数据位都处理完

（IV）清除方向标志指令（Clear direction flag）

- 指令格式：CLD
- 指令功能：DF标志位清0
- DF标志清0后，所有串操作指令每执行一次操作会自动使SI与DI中的偏移量增加，指向下一个数据。

（V）方向标志置位指令（Set direction flag）

- 指令格式：STD
- 指令功能：DF标志位置1
- DF标志置1后，所有串操作指令每执行一次操作会自动使SI与DI中的偏移量减少，指向下一个数据。

（VI）清除中断标志指令（Clear interrupt-enable flag）

- 指令格式：CLI
- 指令功能： IF标志位清0，关闭CPU处理可屏蔽中断的硬件开关。

（VII）中断标志置位指令（Set interrupt-enable flag）

- 指令格式：STI
- 指令功能：IF标志位置1，打开CPU处理可屏蔽中断的硬件开关。

标志位操作指令

- 标志位影响：注意，标志位操作指令仅影响该指令特定针对的标志位，对其它标志位不会造成任何影响。

2）与外部事件同步的指令

- 指用于CPU芯片与连接在总线上的其他功能部件保持同步的指令。

- CPU与外部事件同步的指令仅要求了解其功能。

（I）停机指令

- 指令格式：HLT
- 指令功能：停止CPU的运转，执行该指令后CPU就停止了所有操作，类似于“休克”状态。
- 如果需要重新让CPU进入运转状态，必须向CPU发送“RESET”信号，重新冷启动计算机系统。

（II）协处理器指令前缀

- 指令前缀格式：ESC

- 功能：ESC是一个特定的二进制编码，标识程序中的协处理器指令，一旦CPU识别到这个指令前缀，就会把指令发送到协处理器执行。

- 在程序中的协处理器指令都需要加上ESC指令前缀，而且原则上在每一条协处理器指令后面都需要跟上一条WAIT指令来进行同步。

（III）等待指令

- 指令格式：WAIT
- 指令功能：在向8087协处理器传送浮点处理指令后，等待8087的浮点运算结束信号，接收到该信号后CPU才继续执行后面的指令。
- 在程序中，可以出现8086/8088CPU的指令，也可以出现协处理器关于浮点运算的指令。
- 通过WAIT指令CPU可以和协处理器同步，分别按照顺序执行各自的指令。

8087协处理器及其指令系统

- 8087芯片是在8086/8088系统中是一个可选的部件，在主板上安装了该部件后，计算机系统才具有浮点处理的能力，否则只能处理整数。

- 在后期的CPU设计中，协处理器和处理器基本上溶为一体了。

- 协处理器指令超出了本门课程的范围，如果感兴趣，可以在课外查阅相关的技术资料。

（IV）总线封锁指令前缀

- 指令前缀格式：LOCK

- 功能：如果一条指令带上该前缀，那么在执行该指令时，CPU锁定总线，阻止该指令在执行阶段失去总线控制权。

- 一般在某些特殊情况下使用这种指令前缀。

3）空操作指令（No operation）

- 指令格式：NOP

- 指令功能：该指令使CPU执行一次空操作，占用三个节拍，不做任何操作，不影响任何寄存器、内存单元和标志位。

- NOP指令可以用于延时较短的延时程序中，也可以在调试程序时用于覆盖其它指令。

8086/8088指令系统的余下部分

- 数据或地址传送、算术或逻辑运算、CPU控制等指令是8086/8088指令系统中的基础部分，并不是全部。

- 这些指令加上必要的语法要素和程序框架，已经可以构成完整的汇编语言程序，但无分支、循环、子程序等程序结构。

- 关于转移指令、BCD码调整指令、串操作指令、中断指令等较复杂的指令将在后面的课程中陆续介绍。