汇编自主学习版

第二章

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。

(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。

EU:控制器 运算器

BIU：总线控制逻辑 指令队列 地址加法器

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。

指令队列对于指令是不可见的。

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。**

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）。

段基值*16=段基址

无符号数：用原码

有符号数：用补码

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 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。

一． 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。**

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

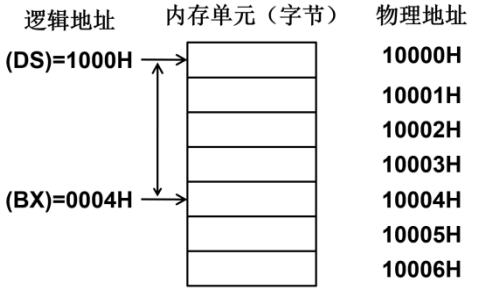
- **代码段**：用于存放程序的机器指令序列；
- **堆栈段**：用于存放程序使用堆栈指令所保存的数据；保存子程序调用指令提供的返回地址；保存中断断点；
  堆栈段的访问一般遵循后进先出（LIFO）原则。
- **数据段**：存放程序需要直接使用的数据，包括变量、数组、字符串等；
- **附加段**：内容不确定，可以根据实际需要决定。通常用于存放串操作指令中的目的串。

二． 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。

- 使用**逻辑地址定位内存单元**的示例：



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；

DS:BX ES:BX

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。
DI 称为**目的变址寄存器**，用于存放偏移量或偏移量分量。
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
SI、BX、DI 用法相似 都是存放偏移量和偏移量分量 SI 叫源变址寄存器，BX 叫基址寄存器

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。

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
CS:IP

三、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**.
（BX 既是基址寄存器又是数据寄存器）
四、 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。**
标志寄存器是实现程序中**分支、循环结构必须的硬件基础**。

状态标志位
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）
1. 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。
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。
CF 针对无符号数
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。
2. 奇偶标志位
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。
偶：1

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

如果 ASCII 码中有奇数个“1”，第 7 位取值为 0；

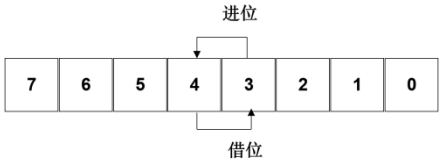
如果 ASCII 码中有偶数个“1”，第 7 位取值为 1；

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。

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。



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。

4 位—0~15 BCD 码表示十进制

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。

4. 零值标志位

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。

5. 符号标志位

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？**）

不，0

SF 标志位位于 FR 的第 7 位。

6. 溢出标志位

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。

用补码计算！！！

7. 控制标志位

1）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**。

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。

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。

2）IF

中断标志位 IF 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

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

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。

- 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。

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

1. push 堆栈，SP 与 SS 配合使用.
2. POPF 就是把标志寄存器的数据出栈，出到标志寄存器里
3. Mul 指令：两个相乘数，要么都是 8 位，要么都是 16 位。8 位乘法/16 位乘法。如果是 8 位，一个数字默认存放在 al 中，另外一个数字存放在其他 8 位寄存器中或者字节型内存单元中。如果是 16 位，一个数字默认存放在 ax 中，另外一个数字存放在其他 16 位寄存器中或者字型内存单元中

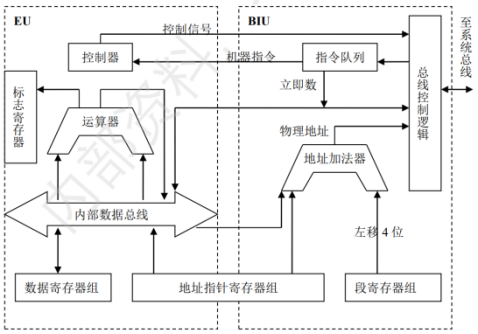
寄存器的特定使用

- 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。

4．1 8086/8088CPU 基本结构与工作原理

8086、8088 两种 CPU 芯片的区别主要在于数据引脚的数量，也即它们所连接的数据总线宽度。



8086、8088 两种 CPU 芯片的区别主要在于数据引脚的数量，也即它们所连接的数据总线宽度。

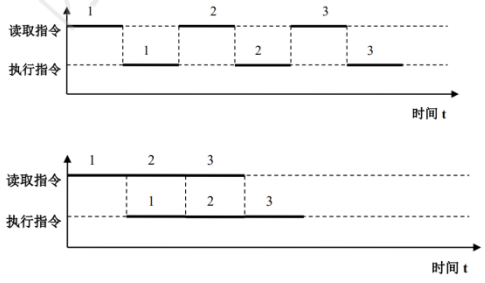
8086 能够连接 16 位数据总线，一次总线操作最多可以读取一个字（在 8086/8088 中为 16 位）的数据量；而 8088 只能连接 8 位数据总线，一次总线操作最多只能读取一个字节(8 位)。

CPU 内部结构分为 BIU 和 EU 两个功能模块。其中 BIU 的全称为 Bus Interface Unit，即总线接口单元，其主要功能是实现 CPU 与系统总线的信号连接、数据交换。所有与总线操作相关的时序过程都由 BIU 模块执行，其中最频繁的是内存单元、端口的数据读写操作，所有有关物理地址的计算机过程也由 BIU 完成

EU 的全称为 Execute Unit，即执行单

元，它的主要功能为解释并执行指令，将CPU分为两个功能模块的目的在于使读取指令与执行指令这两个步骤能够在时间上重叠，从而实现基本的流水线结构，这样，执行当前指令的时间与读取下一条指令的时间就可以重叠，而加快整个程序的执行速度。当然，在时间上使两个步骤重叠的必要条件是当前指令不使用总线操作。换言之，有时这两个步骤并不能在时间上重叠

我们可以通过一个实际例子来体会流水线结构加快程序执行速度的功能。假设有一个由3条指令组成的程序片段，并且假设这3条指令都不使用总线操作，我们通过串行读取/执行指令的方式和EU/BIU流水线方式分别来执行这段程序。那么，这两种方式在读取/执行指令方面消耗的时间分别如图4.1.2（a）和4.1.2（b）。



（b）EU/BIU 流水线方式消耗的时间

图 4.1.2 串行方式与流水线方式连续执行3条指令时的耗时对比

在流水线方式中，使执行当前指令的时间与从内存读取下一条指令的时间重叠，因而大幅提高了程序的执行速度，程序中包含的指令越多，则优势越突出。我们将CPU启动总线进行操作所耗费的时间称为总线周期，将CPU启动总线读取机器指令所耗费的时间称为取指周期，将执行机器指令所耗费的时间称为执行周期。

指令队列：

指令队列这一部件位于BIU模块，它具有多个字节的存储单元。每次BIU模块启动读取下一条指令的操作后，将所读取到的指令（字节数随不同指令而变化）暂存在指令队列中。只要总线出现空闲，BIU模块就会自动启动读取指令的时序操作，直到指令队列填充满为止。 位于 EU 中的控制器会将指令队列中的机器指令操作码字段读取出来，并译码生成相应控制信号，从而执行指令。

指令队列中的机器指令被读取的顺序按照先进先出（FIFO）的队列结构原则，最先由 BIU 保存在指令队列中的指令，最先被 EU 读出执行。

读取指令这一时序过程与执行指令无关，它是由 BIU 自动启动的因而指令队列与我们后面将要介绍的寄存器组不同，它们不能被机器指令访问。换言之，我们在设计汇编语言程序时，不能操作指令队列，它是由 CPU 自动管理的。

控制器：

控制器位于 CPU 的 EU 模块，它是 CPU 的控制中心，无论是作用于CPU内部的控制信号，还是CPU发送到系统总线的控制信号，几乎全部在这里产生，而产生控制信号的依据则来源于对机器指令操作码字段的译码。

地址加法器：

地址加法器位于 BIU 模块，它的功能是由逻辑地址计算物理地址，并最终由该器件将物理地址传送到 BIU 的总线控制逻辑，并由总线控制逻辑将物理地址传送到地址总线。8086/8088 CPU 能支持的地址总线宽度为 20 位， 对于内部存储器而言，其寻址范围为 1M 字节，而访问端口时仅使用地址总线的低 16 位， 因而端口寻址范围为64K 字节。为了协调 CPU 内外地址位数的差异，8086 / 8088 CPU 采用了对存储器的分段管理方式，并使用逻辑地址为 CPU 内部的物理地址表达形式。

逻辑地址由两个16位的地址分量构成，其中一个为段基值，另一个为偏移量，两个分量均为无符号数编码。

$$段基址 = 段基值 \times 16$$
$$物理地址 = 段基址 + 偏移量$$

偏移量的含义为距离段基址的偏移字节数。偏移量指定不同，那么所访问的内存单元也就不同。

任意指定一个确定的逻辑地址，那么经过转换后得到的物理地址是唯一的。换言之，给定一个逻辑地址，就能唯一的定位一个内存单元。但是相反的转换过程并不能得到唯一的结果，给定一个物理地址，我们并不能得到唯一的段基值、偏移量分解。我们在进行汇编语言程序设计时所使用的内存单元地址也就是逻辑地址，而非物理地址。物理地址仅在 CPU 执行总线操作时出现，换言

之，物理地址存在于程序的执行阶段，而非程序的设计阶段，对端口地址而言，由于其仅使用地址总线的低 16 位，因此，逻辑地址等同于物理地址，我们不去区分它们。

地址加法器所完成的操作就是将逻辑地址转换为物理地址后，提交给总线控制逻辑。这样的地址转换可能发生在 BIU 读取指令时，也可能发声在 EU 执行指令时，还可能发生在其它的时序过程中。

运算器：

运算器位于 CPU 的 EU 模块，用于完成机器指令要求的算术运算、逻辑运算功能，是 CPU 内的运算中心，最大支持双操作数运算，例如两个操作数相加或相减等。它还可用于生成逻辑地址中的偏移量分量。若为数据运算，其操作数可能来源于数据寄存器、内存单元、指令中的立即数；若为偏移量运算，其操作数均解释为偏移量分量，来源于地址指针寄存器、指令中的位移量。

总线控制逻辑：

总线控制逻辑为于 BIU 模块，用于控制在适当的时刻向总线传递相应信号，或从总线上接收相应信号，CPU 只能通过总线控制逻辑才能与系统总线实现信号交互。由于总线控制逻辑是 CPU 自动进行管理的，我们的指令没办法直接操作该器件。

数据寄存器：

数据寄存器是在机器指令执行时，被机器指令所使用的寄存器。AX、BX、CX、DX。AX 的全称为累加器，BX 为基址寄存器，CX 为计数寄存器，DX 为数据寄存器。

	15 8	7 0
AX	AH	AL
BX	BH	BL
CX	CH	CL
DX	DH	DL

这里还需说明一点，同一个寄存器可能属于不同的寄存器组，例如 BX 寄存器，它既属于数据寄存器组，也属于地址指针寄存器组，后面我们还将介绍它作为地址指针寄存器的用途。

段寄存器：

4 个段寄存器分别为 CS、DS、SS、ES，既代码段段寄存器、数据段段寄存器、堆栈段段寄存器、附加段段寄存器。4 个段寄存器分别用于指示 4 种段在内存中的起始地址，一个段对应内存中一块连续的存储区域。代码段用于存放由机器指令组成的程序；数据段通常用于保存变量数据；堆栈段用于保存中断断点、子程序返回点、用户使用堆栈临时保存的数据等；附加段通常用于串操作，也可以根据程序员的设计而具有其它功能。虽然 4 种类型的段在源程序中都分别可以定义多个，但每种类型的段在任意时刻仅有一个可以被 CPU 访问，那就是在相应段寄存器中保存了段基值的当前段。

地址指针寄存器：

地址指针寄存器用于提供内存单元逻辑地址中的偏移量或构成偏移量的分量。地址指针寄存器总共包括 5 个 16 位寄存器，分别为 BX、SP、BP、SI、DI，它们的全称分别为基址寄存器、堆栈指针寄存器、基址指针寄存器、源变址寄存器、目的变址寄存器。既然地址指针寄存器提供的是偏移量或其分量，那么它们一定具有搭配使用的段寄存器，因为只有这样才能构成完整的逻辑地址。

地址指针寄存器	与段寄存器的搭配关系
BX	默认与 DS 搭配，但在程序设计时可以修改，可与任意段寄存器搭配
SP	只能与 SS 搭配
BP	默认与 SS 搭配，但在程序设计时可以修改，可与任意段寄存器搭配
SI	默认与 DS 搭配，但在程序设计时可以修改，可与任意段寄存器搭配
DI	一般指令中默认与 DS 搭配，但在程序设计时可以修改，可与任意段寄存器搭配；在串操作指令中只能与 ES 搭配

控制寄存器：

控制寄存器包括指令指针寄存器 IP 和标志寄存器 FR，控制寄存器是指能够直接或间接控制程序执行流程的寄存器。IP 寄存器的英文全称为 Instruction Pointer，即指令指针，其含义非常明确，因为该寄存器中保存的是 CPU 下一条将要从内存中读取的指令在当前代码段（CS 指向的段）中的首字节偏移量，IP 固定与 CS 搭配使用，形成下一条即将被读取指令的逻辑地址。CPU 中的 BIU 模块每次从内存中读取一条机器指令后，都会自动修改 IP 中的偏移量，使之指向下一条机器指令的首字节。换言之，当执行某条机器指令时，IP 中的偏移

量已经指向下一条指令。从上述分析看来，IP 寄存器的属性本来更接近于上一小节的地址指针寄存器。但是转移指令、循环控制指令、子程序调用与返回等指令都是通过修改 IP 中的偏移量来达到控制程序流程的目的。以便实现程序中的分支、循环结构以及子程序的调用与返回等功能。因此，由于修改指令指针能够控制程序流程，这里将它归类到控制寄存器。

FR：该寄存器为 16 位寄存器，标志寄存器中的有效的标志位共有9位，其中，CF、PF、AF、ZF、SF、OF 为六个状态标志，用于反映最近一次影响标志位的算术或逻辑运算中，运算过程、运算结果的一些性质；TF、IF、DF 为三个控制标志，用于控制 CPU 对某些特定事件的处理方式，以及控制 CPU 的工作模式。多数条件转移指令和部分循环控制指令会根据某些状态标志的取值来确定是否改变程序的执行流程，换言之，FR 的状态标志能够间接影响程序执行的流程，因此，FR 被归类到控制寄存器。

标志位：

（1） 进位标志位（CF）

执行加、减算术运算指令时，如果最高位（字节为第 7 位，字为第 15 位）产生进位或借位，那么 CF 标志将被置 1，否则 CF 标志被清 0，

适用于无符号数

执行移位指令时，按照顺序被移出的最后一位保存在 CF 中

（2） 奇偶标志位（PF）

如果运算结果的低 8 位中包含偶数个为“1”的数据位，PF 将被置 1，如果包含奇数个“1”，PF 将被清 0。

偶 1 奇 0

（3） 辅助进位标志位（AF）

执行加、减算术运算指令时，如果第 3 位（最低位为第 0 位）产生进位或借位，则 AF 标志被置 1，否则 AF 标志被清 0。AF 标志的产生规则与 CF 一致。

（4） 零值标志位（ZF）

当执行算术运算或逻辑运算指令时，如果运算结果为 0，则 ZF 标志被置 1，否则 ZF 标志被置 0。ZF 标志的用途非常广泛，例如，比较两个编码是否相等、判断计数是否为 0、判断存储单元中指定位的取值等。

（5） 符号标志位（SF）

当执行算术运算或逻辑运算指令后，SF 标志与运算结果的最高位保持一致，当程序员将参加运算的操作数解释为补码时，SF 可以看作运算结果的符号位。但应当注意，当程序员未将操作数解释为补码时，SF 无意义，SF 标志位在条件转移指令中通常与 OF 标志配合使用，以判断两个补码间的大小关系。

（6） 溢出标志位（OF）

当执行算术运算指令后，如果按照补码解释的运算出现溢出，则 OF 标志被置 1，如果按照补码解释的运算没有溢出，则 OF 标志被清 0。

看作有符号数。

OF 标志可用于判断补码运算是否溢出，也可与 SF、ZF 配合使用，以判断两个补码间的大小关系。

（7） 单步跟踪标志位（TF）

TF 标志位为控制标志位，当它被清 0 时，CPU 工作于连续执行指令的工作模式下，它会不断从内存中读取机器指令，并不断地解释和执行它们；当 TF 标志被置 1 时，CPU 工作于单步模式下，每执行完当前指令后，CPU 内部就会产生一次单步中断，将当前 CPU 中寄存器、标志位的取值都在屏幕上显示出来，并暂停执行。

（8） 中断使能标志位（IF）

IF 标志位为控制标志位，当它被清 0 时，CPU 不会响应任何可屏蔽中断，但仍然会响应不可屏蔽中断；当它被置 1 时，CPU 会响应所有可屏蔽中断和不可屏蔽中断。

（9） 方向标志位（DF）

DF 标志位为控制标志位，用于控制

串操作指令的操作方向。当它被清 0 时，串操作指令按照地址递增的方向进行操作；当它被置 1 时，串操作指令按照地址递减的方向进行操作。

标志位的有效性：

（1）首先，我们所关心的运算指令是否会影响我们判断的标志位？

不同的指令对标志位的影响是不同的，在程序设计时需要注意，所使用的指令是否会影响我们判断的标志位。例如， INC 指令的功能为把指定存储单元中的数据加 1，但是该指令并不影响 CF 标志，因此我们根据 CF 标志来判断 INC 指令的执行结果就是错误的。

（2）其次，我们所关心的指令是否对标志位形成了有意义的影响？

例如 AND 指令，其功能为逻辑与操作，它要影响 AF 标志，但无任何意义。对这种无意义的标志位影响也需要特别注意，因为这些指令确实改变了相应标志位，只是对这种影响不作任何有意义的解释，对我们 的程序设计也没有任何帮助。

（3）最后，程序设计人员对操作数的解释是否与标志位的含义相符？

部分标志位能够被有效使用是有操作数假定的。

寄存器的隐含使用与特定使用：

（1）PUSH AX

该指令将（AX）压栈，隐含使用（SP）作为指向堆栈栈顶的偏移量，压栈操作完会被减 2。

（2）MUL BL

该指令完成无符号数编码的乘法操作，将（BL）与（AL）相乘，乘法结果保存该指令对 AL、AX 寄存器的使用都是隐含使用方式。

例 4．2．2 下列汇编指令中存在寄存器的特定使用。

（1）IN AL, 60H

该指令读取 60H 号端口中的 8 位数据，并保存到 AL 寄存器，这里 AL 寄存器的使用属于特定使用。

（2）SHL AX，CL

该指令对（AX）实施逻辑左移操作，左移的位数有（CL）给出，这里 CL 寄于特定使用。

第五章 基本指令系统

汇编指令基本格式：

构成汇编指令的基本元素包括操作助记符与操作数（或操作数地址）。操作助记符指明指令的功能，而操作数指明指令操作的数 据，尽管我们把它称为“操作数”，但它实质上是某种编码，而非通常意义下的数。8086/8088 CPU 的指令可以分为双操作数指令、单操作数指令、 无操作数指令三类。

（1） 双操作数指令

（1）双操作数指令

MOV	AX，	BX
操作助记符	目的操作数（地址）	源操作数（地址）

操作助记符在前，目的操作数在中间，源操作数在最后，源操作数是指提供给指令作为原始数据的操作数，目的操作数则是指操作完成后的结果数据。

（2） 单操作数指令

NOT	AX
操作助记符	源、目的操作数

在本例 中，NOT 指令的功能为逻辑非操作，它会将（AX）按位取反，然后保存回 AX 寄存器。因此，在本例中，AX 既提供原始数据，又作为保存运算结果的地址，充当了两个角色。

（3） 无操作数指令

NOP

操作助记符

作为一条完整的指令，操作助记符是必不可少的构成元素。

注意事项：

i. 指令中明确给出的操作数数量不一定与指令实际使用的操作数数量相符。因为在指令中可能出现某个操作数可能

同时具备源、目的操作数双重含义，因而这种操作数实质上应理解为两个操作数，只是它们在物理上重叠罢了。并且，指令中可能出现对操作数的隐含使用。因此，指令中实际的操作数数量大于等于指令中明确给出的操作数数量。

ii. 双操作数指令中仅允许最多一个操作数为内存单元或端口，不能两个操作数同为内存单元或同为端口。

iii. 单操作数指令中，操作数不能使用立即数，只能使用寄存器、内存单元。

寻址方式：

1. 寄存器寻址

如果操作数在寄存器中，则指令中以寄存器名称(地址)的形式给出，并且称该操作数的寻址方式为寄存器寻址方式。寄存器寻址方式是所有寻址方式中速度最快的。

例 5. 2. 1 以下指令中的源操作数、目的操作数均为寄存器寻址方式。

```
MOV  AX，BX      ；（BX）保存至 AX
ADD  AL，DL      ；（AL）与（DL）相加后，运算结果保存至 AL
SUB  BX，CX      ；（BX）减去（CX），运算结果保存至 BX
```

2. 立即数寻址

如果操作数包含在机器指令内，以立即数字段的形式体现，则称该操作数为立即数寻址方式。立即数寻址方式仅能用于源操作数寻址，因为立即数是操作数本身，而不是地址，不能用于保存目的操作数。并且，在单操作数指令中不能使用立即数寻址。当 BIU 模块启动取指周期从内存读取机器指令时，立即数就随同机器指令被保存在 BIU 的指令队列中，EU 模块执行指令时，只需从指令队列中取得立即数，而不需要单独启动总线操作来读取。在部分情况下，立即数的获取可以认为是在 CPU 内部完成的。因此，立即数寻址方式的速度仅次于寄存器寻址方式，但比其它需要在指令执行周期中启动总线操作的寻址方式要快。

```
MOV  AL，25
```
源操作数为立即数寻址，目的操作数为寄存器寻址，该指令将立即数 25 传送至 AL 保存。

```
ADD  BL，42
```
源操作数为立即数寻址，目的操作数为寄存器寻址，该指令完成（BL）+42 的运算，运算结果保存至 BL 寄存器。

3. 存储器寻址

如果操作数位于内存单元中，则称该操作数为存储器寻址。读者应注意存储器寻址与立即数寻址的区别。最初立即数与存储器寻址的操作数都在内存单元中，但立即数是随机器指令一起在 BIU 的取指周期读取到 CPU 内部的，而存储器寻址却发生在指令的执行周期，需要在执行周期内单独启动总线操作来完成。存储器寻址比寄存器寻址、立即数寻址速度都要慢。

偏移量可以由多种方式来获取，不同的偏移量获取方式则形成了不同的存储器寻址方式。内存单元的偏移量可以看作是由三种偏移量分量组合相加得到的，这三种分量分别为机器指令位移量字段提供的位移量、基址寄存器提供的基址分量、变址寄存器提供的变址分量。不同的分量组合就形成不同的存储器寻址。偏移量也被称为有效地址，简称 EA。

（1） 直接寻址

直接使用机器指令中的位移量字段作为内存操作数的偏移量，则称该操作数为直接寻址方式。

$$EA = DISP$$

DISP 表示位移量，即 disparity，仅在直接寻址方式中，偏移量才等于位移量。在程序设计中，偏移量始终被看作无符号数编码，但位移量既可以看作无符

号数编码，也可看作补码。

例 5．2．3 以下指令中使用了直接寻址方式

MOV　AL，[1000H]

该指令源操作数为直接寻址方式，目的操作数为寄存器寻址方式，方括号内为十六进制的数值**位移量**（注意在语法上与立即数的区别），它将直接作为偏移量，如果没有指定段前缀，则默认使用（DS）为段基值。该指令将逻辑地址为（DS）：1000H 的字节单元数据从总线读取出来，并保存在 AL 中。

MOV　ES：[0100H]，BX

该指令源操作数为寄存器寻址，目的操作数为直接寻址，指明了段前缀 ES，即指明使用（ES）作为段基值。该指令将（BX）通过总线写入字内存单元（16 位），字内存单元的逻辑地址为（ES）：0100H。

MOV　VAR1，BL

该指令源操作数为寄存器寻址，目的操作数为直接寻址，指令中所给位移量是一个变量名称，其实质为**符号位移量**，在汇编后，符号位移量会被替换为数值位移量，二者本质上是相同的。直接寻址方式中的符号位移量默认使用（DS）作为段基值。

MOV　VAR1+2，AL

该指令源操作数为寄存器寻址，目的操作数为直接寻址，变量名称后加了一个常量 2，表示 EA 为 VAR1 对应的数值位移量加 2，该加法运算在汇编阶段完成，而不是指令执行阶段，因此寻址方式仍然属于直接寻址。

（2）　间接寻址

如果内存操作数的偏移量由地址指针寄存器 BX、BP、SI、DI 其中之一提供，则称该操作数为寄存器间接寻址方式。其中，（BX）、（BP)称为基址分量,(SI)、（DI）称为变址分量。

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

例 5．2．4 以下指令中都使用了寄存器间接寻址方式

ADD　CL，[BX]

该指令的目的操作数为寄存器寻址，源操作数为寄存器间接寻址，使用（BX）基址分量直接作为偏移量，默认使用（DS）作为段基值。该指令将（BX）指示的字节内存单元数据与（CL）相加，运算结果保存在 CL 寄存器。

SUB　ES：[SI]，AX

该指令的源操作数为寄存器寻址，目的操作数为寄存器间接寻址，使用（SI）变址分量直接作为偏移量，使用（ES）作为段基值。该指令用（SI）指示的字内存单元数据减去（AX）运算结果保存在字内存单元中。

在寄存器间接寻址方式中，BX、SI、DI 默认与 DS 搭配，BP 默认与 SS 搭配。

（3）　基址寻址与变址寻址

如果内存操作数的偏移量由基址分量与位移量分量相加得到，则称该操作数为基址寻址方式；如果内存操作数的偏移量由变址分量与位移量分量相加得到，则称该操作数为变址寻址方式。

基址寻址的 EA 计算：

$$EA=(BX)+DISP$$
$$EA=(BP)+DISP$$

变址寻址的 EA 计算：

$$EA=(SI)+DISP$$
$$EA=(DI)+DISP$$

例 5．2．5 下列指令中部分使用了基址寻址，部分使用了变址寻址

MOV　AL，[BP]100H

该指令的目的操作数为寄存器寻址，源操作数为基址寻址，EA=（BP）+0100H，默认使用（SS）作为段基值。

MOV　BYTE PTR [SI]220H，30H

该指令的源操作数为立即数寻址，目的操作数为变址寻址，EA=（SI)+0220H，默认使用（DS）作为段基值。注意，“BYTE PTR”指明内存单元类型为字节类型，在无寄存器寻址的指令中，需指明内存单元的类型，否则数据类型将出现二义性，汇编时将出现一个警告错误。

读者应注意，在基址和变址寻址中，BX、SI、DI 默认与 DS 搭配，而 BP 仍然默认与 SS 搭配。

（4）　基址变址寻址

如果内存操作数的偏移量由基址分量、变址分量、位移量分量三种分量相加得到，那么称该操作数为基址变址寻址。其中，基址分量由（BX）或（BP）提供，变址分量由（SI）或（DI）提供。

$$EA=\begin{Bmatrix}BX\\BP\end{Bmatrix}+\begin{Bmatrix}SI\\DI\end{Bmatrix}+DISP$$

AND　AL，[BX][SI]0020H

该指令的目的操作数为寄存器寻址，源操作数为基址变址寻址，EA=（BX）+（SI）+0020H，默认使用（DS）为段基值。该指令将字节内存单元数据与（AL）作逻辑与运算，运算结果保存在 AL 中。

OR　[BP][DI]0010H，BL

该指令的源操作数为寄存器寻址，目的操作数为基址变址寻址，EA=（BP）+（DI）+0010H，默认使用（SS）作为段基值。该指令将（BL）与字节内存单元数据作逻辑或运算，运算结果保存在内存单元中。

在基址变址寻址中，默认的段寄存器搭配由基址寄存器决定，BX 默认与 DS 搭

配，BP 默认与 SS 搭配。

4. 其他寻址方式

1） 串操作寻址方式

只要所使用的指令为串操作指令，则内存操作数的寻址方式均为串操作寻址方式。

2） 端口寻址方式

能够访问端口的指令只有两种，in 指令用于读端口，out 指令用于写端口。在端口读写指令中，位于端口中的操作数均为端口寻址方式，其中又可分为直接端口寻址方式和间接端口寻址方式。

3） 隐含寻址方式

如果操作数在指令中没有明确给出，但指令隐含地使用了它，那么这些隐含操作数的寻址方式统称为隐含寻址方式。

普通的寻址方式都是由机器指令的寻址方式字段来指定，但隐含寻址方式却是由机器指令的操作码字段来确定的。换言之，指令的功能确定其对应的隐含寻址。

Eg:

（6）AND VAR1+4，DL

源操作数：寄存器寻址

目的操作数：直接寻址，EA=VAR1+4，使用 DS 段寄存器

（7）PUSHF

源操作数、目的操作数均为隐含寻址

（9）ADC BYTE PTR [BP][SI]0210H，45H

源操作数：立即数寻址

目的操作数：基址变址寻址，EA=(BP)+(SI)+0210H，使用 SS 段寄存器

（10）OR ARRY[BX][DI]，CL

源操作数：寄存器寻址

目的操作数：基址变址寻址，EA=(BX)+(DI)+ARRY，使用 DS 段寄存器

基本指令系统

按照指令的功能分类， 8086/8088 指令可分为传送类指令、算术运算类指令、位操作类指令、程序转移类指令、串操作类指令、处理器控制指令。

1. 传送类指令

1） 数据传送指令

指令格式：MOV DEST，SRC

指令功能：$(SRC) \Rightarrow DEST$ 或 $SRC \Rightarrow DEST$

标志位影响：无

若源操作数为地址形式，则将地址所指示的存储单元数据传送至目的操作数保存，若源操作数为立即数，则将立即数传送至目的操作数保存。

应当注意，双操作数指令中只能有一个内存操作数出现，若要实现两个内存单元的数据传送，则需要使用两条 MOV 指令，并通过通用寄存器作中转才能实现

将字节内存单元 VA1 中的数据传送至字节内存单元 VA2 中保存。

```
MOV  AL，VA1
MOV  VA2，AL
```

MOV 指令不能对段寄存器直接传送立即数，需要通过通用寄存器中转，并且 MOV 指令也不能直接在两个段寄存器之间直接传递数据，需要通过通用寄存器中转。

（6）MOV DS，ES

错误，段寄存器间不能使用 MOV 指令直接传递数据，必须通过通用寄存器作为中转

（7）MOV DS，0620H

错误，使用 MOV 指令向段寄存器传递数据时，不能使用立即数

例 5．3．2 将立即数 1100H 传送至段寄存器 DS、ES 保存。

```
MOV  AX，1100H
MOV  DS，AX
MOV  ES，AX
```

2） 交换指令

指令格式：XCHG DEST, SRC

指令功能：$(SRC) \Rightarrow DEST$，$(DEST) \Rightarrow SRC$（两个操作同时完成）

标志位影响：无

该指令将源操作数与目的操作数交换保存，指令中不能使用立即数，也不能使用段寄存器，与其它双操作数指令一样，最多只能出现一个内存操作数。

例 5．3．3 交换字节内存单元 VA1 与字节内存单元 VA2 中的数据。

```
MOV   AL，VA1
XCHG  AL，VA2
MOV   VA1，AL
```

3） 取标志位指令

指令格式：LAHF

指令功能：$(FR)_{0\sim7} \Rightarrow AH$

标志位影响：无

该指令将标志寄存器低 8 位数据传送到 AH 保存。读者应注意，该指令是无操作数指令，但实质上具有 AH、标志寄存器两个操作数，对 AH 寄存器、标志寄存器的寻址方式均为隐含寻址方式。

4） 存标志位指令

指令格式：SAHF

指令功能：$(AH) \Rightarrow FR_{0\sim7}$

标志位影响：标志寄存器低 8 位，包括 SF、ZF、AF、PF、CF

该指令的功能与 LAHF 正好相反，将（AH）传送到标志寄存器低 8 位保存，由于标志寄存器是该指令的隐含目的操作数，因此位于低 8 位范围的 5 个状态标志将受到影响。

5） 入栈指令

指令格式：PUSH SRC

指令功能：
$$(SP)-2 \Rightarrow SP$$
$$(SRC) \Rightarrow (SP)$$

标志位影响：无

该指令先将（SP）减 2，使之指向一个空的栈顶，再将源操作数传送至栈顶保存。该指令为单操作数寻址，源操作数必须为 16 位（字类型）的寄存器或内存单元，不能为 8 位存储单元或立即数。但该指令实质上有两个操作数，除源操作数外，隐含使用了（SP）指向的栈顶内存单元作为目的操作数。在程序设计中，如果需要在堆栈中临时保存数据，可以使用 PUSH 指令实现。

（1）PUSH 8243H

错误，单操作数指令不能使用立即数

6） 出栈指令

指令格式：POP DEST

指令功能：
$$((SP)) \Rightarrow DEST$$
$$(SP)+2 \Rightarrow SP$$

标志位影响：无

出栈指令的功能与入栈指令相反，它先将栈顶字单元中的数据传送至目的操作数保存，然后将（SP）加 2，以丢弃出栈数据并修改栈顶。指令功能表达式中（（SP））表示由（SP）指示的栈顶内存单元的内容。出栈指令对操作数的要求与入栈指令一致，且目的操作数只能是字类型。入栈、出栈操作均遵循堆栈结构的后进先出原则（LIFO），即 最先入栈的数据最后才能出栈。

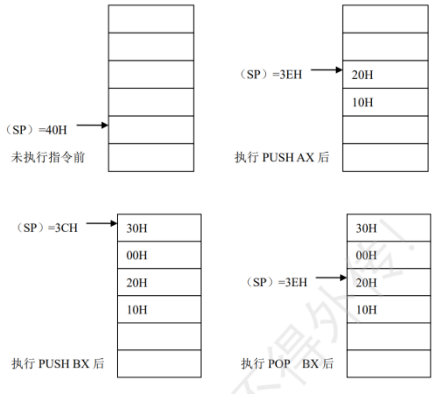
（2）POP AL

错误，进栈、出栈指令的操作数应为 16 位

AL 仅有八位

例 5．3．4 假设（AX）=1020H，（BX）=0030H，（SP）=0040H，下列程序片段先将（AX）、（BX）入栈保存，然后再出栈恢复，通过图示可以观察到堆栈的变化。

```
PUSH  AX
PUSH  BX
POP   BX
POP   AX
```



7） 标志压栈指令

指令格式：PUSHF

指令功能：$(SP)-2\Rightarrow SP$
$(FR)\Rightarrow (SP)$

标志位影响：无

PUSHF 指令先修改（SP），使之指向空栈顶，然后将 16 位标志寄存器中的数据传送至栈顶保存。PUSHF 指令为无操作数指令，但其中有两个隐含操作数，隐含的源操作数为标志寄存器，隐含的目的操作数为栈顶内存单元。

8）标志出栈指令

指令格式：POPF

指令功能：$((SP))\Rightarrow FR$
$(SP)+2\Rightarrow SP$

标志位影响：全部标志位

POPF 指令的功能与 PUSHF 指令相反，它先将（SP）指示的栈顶字内存单元数据传送至 16 位标志寄存器保存，然后将（SP）加 2，以丢弃已出栈数据并修改栈顶位置。POPF 为无操作数指令，具有两个隐含操作数，源操作数为栈顶内存单元，目的操作数为 16 位标志寄存器。

9）装入有效地址指令

指令格式：LEA DEST，SRC

指令功能：$EA_{SRC}\Rightarrow DEST$

标志位影响：无

该指令将源操作数的有效地址传送至目的操作数保存。由于具有有效地址概念的存储单元只能是内存单元，因此该指令的源操作数只能是内存单元；并且，由于有效地址为 16 位，因此能够存储有效地址的目的操作数只能是 16 位存储单元，又由于双操作数指令中最多允许出现一个内存单元，所以，目的操作数只能是 16 位的通用寄存器。应注意该指令与 MOV 指令的区别，MOV 指令访问源操作数的数据，LEA 指令却是获取源操作数的有效地址。

例 5．3．5 假设（DS）=1000H，字内存单元 1000H：0100H 中的数据为 0820H，试分析下面两条指令的执行结果。

```
MOV  SI，[0100H]
LEA  DI，[0100H]
```

执行第一条指令后，（SI）=0820H，执行第二条指令后，（DI）=0100H。

（8）LEA BX，AX

错误，LEA 指令的源操作数必须为内存单元

10）装入地址指针指令

指令格式：LDS DEST，SRC
LES DEST，SRC

指令功能：$(SRC)\Rightarrow DEST$
$(SRC+2)\Rightarrow DS或ES$

标志位影响：无

装入地址指令包括 LDS、LES 两条指令，源操作数只能是内存单元，目的操作数只能是 16 位通用寄存器，并隐含使用 DS 或 ES 段寄存器作为隐含目的操作数。该指令将源操作数地址指向的字内存单元数据传送至目的操作数保存，将源操作数地址加 2 指向的字内存单元数据传送至 DS 或 ES 保存。装入地址指针指令是将 SRC 源操作数地址指向的双字数据作为地址指针看待，低字解释为有效地址，高字解释为段基值，并将低字传送到 DEST 指定的 16 位通用寄存器保存，高字传送到 DS 或 ES 保存。与 LEA 指令不同，LDS 或 LES 是将内存单元中的数据解释为地址，而不是直接获取内存单元的有效地址。

例 5．3．6 假设（DS）=1000H，字内存单元 1000H：0100H 中的数据为 0820H，字内存单元 1000H：0102H 中的数据为 2000H，试分析下列指令的执行结果。

```
LEA  BX，[0100H]
LDS  BX，[0100H]
```

执行 LEA 指令后，（BX）=0100H，执行 LDS 指令后，（BX）=0820H，（DS）=2000H。

2. 算术运算类指令

1）加法指令

指令格式：ADD DEST，SRC

指令功能：$(SRC)+(DEST)\Rightarrow DEST$ 或 $SRC+(DEST)\Rightarrow DEST$

标志位影响：所有状态标志，包括 OF、SF、ZF、AF、PF、CF

加法指令将源操作数与目的操作数

相加，运算结果保存至目的操作数。源、目的操作数均可使用寄存器、内存单元，但至多允许出现一个内存单元；

Eg :

（5）ADC VAR1，VAR2

错误，8086 指令系统的双操作数指令中，必须有一个是寄存器，不能两个操作数同为内存单元

操作数的类型可以是字节，也可以是字；源操作数可以使用立即数，目的操作数不能使用立即数。加法指令的操作数实际上有三个，因为指令中目的操作数既用于提供原始数据，又用于保存运算结果，即它充当了源、目的操作数的双重角色。

例 5．3．7 假设（AL）=32H，（AH）=0F1H，试分析执行如下加法指令后，各寄存器的内容与各标志位的取值，并对溢出情况加以分析。

ADD AL，AH

这里将操作数转换为二进制来分析其运算过程：

```
  00110010
+ 11110001
1 00100011
```

因此运算结果为 23H，执行该指令后（AL）=23H，（AH）=0F1H，只有目的操作数发生变。下面我们来分析各标志位在运算后的取值。

CF：由于运算时最高位产生了进位，CF=1。

PF：运算结果的低 8 位中共有 3 个为“1”的数据位，“1”数据位个数为奇数

PF 只看低八位

AF：由于运算时第 3 位（最低位为第 0 位）没有向第 4 位进位，AF=0。

ZF：运算结果为非零编码，因此 ZF=0。

SF：运算结果最高位为“0”，因此 SF=0。

OF：如果操作数解释为补码，则为“正+负”的情况，不会出现溢出，OF=0。

下面我们再对溢出情况作出分析：

如果程序员将操作数解释为无符号数编码，由于 CF=1，无符号数运算超出了表达范围，产生了溢出。

如果程序员将操作数解释为补码，由于 OF=0，补码运算没有溢出。

加法指令的二意性，参加运算的操作数既可以解释为无符号数编码，也可以解释为补码。换言之，执行一条加法指令实际上同时完成了两种不同解释的运算，即无符号数编码运算和补码运算，两种运算过程的状态则由两套标志位生成电路分别记载于 CF 和 OF 标志。但是每次对编码运算的解释只能取其中之一。加法指令对标志位的影响之所以如此设计，是为了简化指令系统，否则补码的加法运算和无符号数的加法运算就要设计成两条不同的指令。

（4）ADD [0100H]，64H

错误，目的操作数应使用 PTR 运算符指出类型，否则具有二义性

正确的写法：ADD BYTE PTR [0100H],64H，（或使用 WORD PTR）

2） 带进位加法指令

指令格式：ADC DEST，SRC

指令功能：$(SRC)+(DEST)+(CF)\Rightarrow DEST$ 或 $SRC+(DEST)+(CF)\Rightarrow DEST$

标志位影响：所有状态标志，包括 OF、SF、ZF、AF、PF、CF

带进位加法指令将源操作数、目的操作数、进位标志 CF 相加后，运算结果保存至目的操作数。注意，运算时 CF 加在最低位。ADC 指令对操作数的限制与 ADD 指令一致，对标志位的解释也与 ADD 指令一致。与 ADD 指令不同的是，ADC 指令的功能主要是用于长操作数（长度超过机器字长限制）的加法，它可以将低字（或低字节）数据加法产生的进位反映到高字（或高字节）的加法中，以保证长操作数加法的连贯性、正确性。

例 5．3．8 假设（AX）=08E7H，（BX）=0485H，现在要求完成两个操作数的加法，但为了体现长操作数的运算特征，我们把加法过程拆分为低字节和高字节两个加法操作，分别使用 ADD 指令和 ADC 指令来完成。试完成长操作数的加法，并分析无符号数编码、补码运算的溢出情况。

实现长操作数加法的指令序列如下：

ADD AL，BL

ADC AH，BH

低字节加法过程如下：

```
  11100111
+ 10000101
1 01101100
```

由于最高位产生进位，CF=1

由于“负+负=正”，补码运算溢出，OF=1

高字节加法过程如下：

```
  00001000
+ 00000100
  00000001
0 00001101
```

由于最高位没有产生进位，CF=0

由于“正+正+正=正”，补码运算没有溢出，OF=0

从上面 CF、OF 标志的判断我们可

以看到，低字节运算与高字节运算得出了截然相反的结论。使用低字节运算得到的标志位并不能用于溢出，判断，正确的判断应在整个加法过程完成之后才能进行。换言之，只有 在高字节加法完成后，分别使用 CF、OF 对无符号数、补码的溢出进行判断才是正确的。

3) 加 1 指令

指令格式：INC DEST

指令功能：$(DEST)+1 \Rightarrow DEST$

标志位影响：五个状态标志，包括 OF、SF、ZF、AF、PF

INC 指令将目的操作数加 1 后保存回目的操作数地址，目的操作数兼作源、目的操作数两种角色。对操作数的限制符合一般单操作数指令的限制，即不能使用立即数，可以使用字或字节类型的通用寄存器、内存单元。

INC 指令并不会影响 CF 标志。INC 指令的用途一般为循环结构中进行计数，或在循环结构中修改用于表示数组下标的存储单元，使程序能够逐个访问数组中的元素。

4) 减法指令

指令格式：SUB DEST，SRC

指令功能：$(DEST)-(SRC) \Rightarrow DEST$ 或 $(DEST)-SRC \Rightarrow DEST$

标志位影响：所有状态标志，包括 OF、SF、ZF、AF、PF、CF

减法指令将目的操作数作为被减数，将源操作数作为减数，完成减法操作后，运算结果保存至目的操作数。减法指令对操作数的限制与加法指令相同。与加法指令一样，减法指令本身也是具有二意性的，究竟是完成的是无符号数编码运算还是补码运算，需要由程序员来确定。

例 5. 3. 9 假设（AL）=32H，（AH）=0F1H，试分析执行如下减法指令后，各寄存器的内容与 CF、OF 标志位的取值，并对溢出情况加以分析。

SUB AL，AH

二进制编码的减法过程如下：（运算结果左边的“1”表示有借位产生）

$$\begin{array}{rr} & 00110010 \\ - & 11110001 \\ \hline 1 & 01000001 \end{array}$$

减法指令执行后，（AL）=41H，（AH）=0F1H。

由于最高位产生借位，CF=1，无符号数编码运算溢出。

由于“正-负=正+正=正”，补码运算没有溢出，OF=0。

不能采用补码加法来判断 CF 标志，因为采用带符号数的运算逻辑来判断无符号数标志位，在逻辑上本身就是混淆的。

5) 带借位减法指令

（5）带借位减法指令（Subtraction with Borrow）

指令格式：SBB DEST，SRC

指令功能：$(DEST)-(SRC)-(CF) \Rightarrow DEST$ 或 $(DEST)-SRC-(CF) \Rightarrow DEST$

标志位影响：所有状态标志，包括 OF、SF、ZF、AF、PF、CF

带借位减法指令与带进位加法指令含义相似，同样需要注意，判断 CF 标志只能使用减法，不能使用补码加法。

6) 减 1 指令

指令格式：DEC DEST

指令功能：$(DEST)-1 \Rightarrow DEST$

标志位影响：五个状态标志，包括 OF、SF、ZF、AF、PF

7) 求相反数指令

指令格式：NEG DEST

指令功能：$0-(DEST) \Rightarrow DEST$

标志位影响：所有状态标志，包括 OF、SF、ZF、AF、PF、CF

NEG 指令的操作数通常解释为补码，其功能是求目的操作数中补码所对应的相反数补码，运算结果保存至目的操作数地址。

例 5. 3. 10 假设 VA+2、VA 两个字内存单元中分别存放了一个 32 位补码的高字和低字，要求设计程序片段获取该 32 补码的相反数补码。

```
NEG   VA          低字取反
MOV   AX，0
SBB   AX，VA+2
```

SBB：ax-（va+2）-CF

```
MOV   VA+2，AX
```

8) 比较指令

指令格式：CMP　DEST，SRC

指令功能：$(DEST)-(SRC)$ 或 $(DEST)-SRC$

标志位影响：所有状态标志，包括 OF、SF、ZF、AF、PF、

DEST-SRC

CMP 指令除了不保存运算结果外，其余所有解释与减法指令 SUB 相同。换言之，CMP 指令仅通过两个操作数的减法运算来影响标志位，而对最终运算结果不保存。

首先，我们考虑无符号数编码的大小关系判断。如果两个无符号数编码相减，CF=0，则说明最高位没有产生借位，对应的情况为被减数大于等于减数。如果 CF=1，则说明最高位产生了借位，对应的情况为被减数小于减数，而 CF=1 正是无符号数编码运算溢出的情况。

无论无符号数编码的减法运算有没有溢出，我们都能根据 CF 的状态来判断两个操作数间的大小关系。

I. OF=0，SF=0：根据 SF=0 我们可以知道运算结果的最高位（符号位）为 0，得到的运算结果应为一个正数补码（包括 0），OF=0 说明运算是没有溢出的，那么此正数运算结果也就是正确的。这种情况对应于被减数大于等于减数的情况。

II. OF=1，SF=1：根据 SF=1 我们可以知道运算结果的最高位（符号位）为 1，得到的运算结果应为一个负数补码，OF=1 说明运算出现了溢出，那么此负数运算结果是错误的。因此，我们可以推论，正确的运算结果必然是一个正数补码（包括 0），这种情况对应于被减数大于等于减数的情况。

III. OF=0，SF=1：根据 SF=1 我们可以知道运算结果的最高位（符号位）为 1，得到的运算结果应为一个负数补码，OF=0 说明运算是没有溢出的，那么此负数运算结果也就是正确的。这种情况对应于被减数小于减数的情况。

IV. OF=1，SF=0：根据 SF=0 我们可以知道运算结果的最高位（符号位）为 0，得到的运算结果应为一个正数补码（包括 0），OF=1 说明运算出现了溢出，那么此正数运算结果是错误的。因此，我们可以推论，正确的运算结果必然是一个负数补码，这种情况对应于被减数小于减数的情况。

综上所述，无论补码运算有没有溢出，我们都可以通过 SF ≠ OF 这一条件来得到被减数大于等于减数的结论，并通过 SF = OF 这一条件来得到被减数小于减数的结论。无论减法操作有没有溢出，我们都可以通过 OF、SF 这两个标志位的相等和不等关系来判断补码的大小关系。如果再结合 ZF 标志，我们可以得到更细致的判断结果。

3. 位操作类指令

1） 逻辑与指令

指令格式：AND　DEST，SRC

指令功能：$(SRC)\wedge(DEST)\Rightarrow DEST$ 或 $SRC\wedge(DEST)\Rightarrow DEST$

标志位影响：所有状态标志，包括 OF、SF、ZF、AF、PF、CF，其中 CF、OF 被强行置为 0，AF 不确定，SF、ZF、PF 的解释与算术运算指令保持一致。

AND 指令将源操作数与目的操作数按位相与，并将运算结果保存至目的操作数。注意，标志位说明中，“AF 不确定”的含义是指 AND 指令会影响 AF，但这种影响没有任何含义。

例 5. 3. 11 假设（AL）=01010110B，要求将（AL）的第 1 位（最低位为第 0 位）分离出来。

AND　AL，00000010B

01010110

00000010 ->00000010

按位相与后，（AL）=00000010B，我们能够体会到，（AL）的第 1 位已经被分离出来，其它数据位均被屏蔽。指令中立即数称为取位模板，目的操作数中与取位模板中为“1”的位对应的数据位被取出，而与取位模板中为“0”的位对应的数据位则被屏蔽。这对我们在程序设计中分离状态端口中不同的状态位，然后再分别加以判断提供了方便的手段。

除“取位操作外”，逻辑与指令还可以用于“位清零操作”，即将指定的二进制位清零

例 5. 3. 12 假设（AL）=01010110B，要求将（AL）的第 1 位（最低位为第 0 位）清

AND AL，11111101B

指令中的立即数可称为“清零模板”。

2）逻辑或指令

指令格式：OR DEST，SRC

指令功能：$(SRC)\vee(DEST)\Rightarrow DEST$ 或 $SRC\vee(DEST)\Rightarrow DEST$

标志位影响：所有状态标志，包括 OF、SF、ZF、AF、PF、CF，其中 CF、OF 被强行

0，AF 不确定，SF、ZF、PF 的解释与算术运算指令保持一致。

例 5. 3. 13 假设（AL）=01010110B，要求将（AL）的第 0 位（最低位为第 0 位）

OR AL，00000001B

指令中的立即数可称为“置位模板”。

3）逻辑异或指令

指令格式：XOR DEST，SRC

指令功能：$(SRC)\oplus(DEST)\Rightarrow DEST$ 或 $SRC\oplus(DEST)\Rightarrow DEST$

标志位影响：所有状态标志，包括 OF、SF、ZF、AF、PF、CF，其中 CF、OF 被强行

0，AF 不确定，SF、ZF、PF 的解释与算术运算指令保持一致。

异或操作的含义为：如果两个相异或的数据位相等，则异或结果为 0；如果两个相异或的数据位不等，则异或结果为 1。

除了完成通常的异或运算外，异或指令还可以用于“位变反操作”，即将指定的二进制位变反。

例 5. 3. 14 假设（AL）=01010110B，要求将（AL）的第 7 位（最低位为第 0 位）变反。

XOR AL，10000000B

指令中的立即数可称为“变反模板”

4）逻辑非指令

指令格式：NOT DEST

指令功能：$\neg(DEST)\Rightarrow DEST$

标志位影响：无

NOT 指令不影响任何标志位。

5）测试指令

指令格式：TEST DEST，SRC

指令功能：$(SRC)\wedge(DEST)$ 或 $SRC\wedge(DEST)$

标志位影响：所有状态标志，包括 OF、SF、ZF、AF、PF、CF，其中 CF、OF 被强行置为 0，AF 不确定，SF、ZF、PF 的解释与算术运算指令保持一致。

除不保存运算结果外，TEST 指令的功能与标志位解释都和 AND 指令完全一致。

例 5. 3. 15 假设（AL）=01010110B，要求测试（AL）的第 1 位（最低位为第 0 位）的取值。

TEST AL，00000010B

运算结果的第 1 位为“0”与否等价于整个运算结果为零与否。如果（AL）中第 1 位为 1，那么逻辑与运算的运算结果非零，ZF=0；如果（AL）中第 1 位为 0，那么逻辑与运算的运算结果为零，ZF=1。如此，我们就可以根据 ZF 的状态来判断（AL）中指定二进制位的取值，并实现相应的分支或循环结构了。

TEST 指令通常可用于测试状态端口中指定状态位的状态，并结合条件转移指令根据当前状态实现分支或循环处理；TEST 指令也可用于对寄存器、内存单元中的指定据位进行测试，根据不同的取值作不同的处理

6）算术左移指令

指令格式：SAL DEST，COUNT

COUNT 的含义为移位位数，如果 COUNT 为 1 时直接用“1”表示，如果 COUNT>1 则只能以 CL 寄存器表示，以（CL）给出移位位数。

指令功能：将（DEST）左移 COUNT 位，从左边移出的最低一位保存到 CF 中，右边空出的数据位则补充 0，其含义为补码乘以 2^{COUNT}。

注意： 1 CF 取值

2 数值*2 的 count 次方

标志位影响：所有状态标志，包括 OF、SF、ZF、AF、PF、CF，其中 AF 标志不确定，其余标志位的解释与算术运算指令一致，但仅当 COUNT=1 时，OF 标志才有意义。

SAL 指令实际上完成的是一种算术运算，因为目的操作数每左移 1 位，相当于乘以 2，

则将各二进制位的权值升高 2 倍，左移 COUNT 位，相当于乘以 COUNT 次 2，这种多次乘 2 的运算可以通过目的操作数不断自加来实现，每次自加则相当于乘以 2，因此,CF 标志仍可理解为目的操作数多次自加后的进位标志。SAL 指令将目的操作数解释为补码。它没有二意性，完成的是补码乘以 2 的 COUNT 次方的操作，可由 OF 来判断运算是否溢出。

但应注意，由于移位操作在硬件上是一次完成的，即便 COUNT>1 时也是如此，因此，当 COUNT>1 时，可能出现符号位多次变化在一次移位操作中发生的情况，但 CPU 中的逻辑电路无法检测到这一情况，最终导致 OF 仅在 COUNT=1 时才能提供有效的标志信息。

如果 COUNT=1，并且移位前后（DEST）的符号位发生了变化，则 CPU 认为补码自加溢出，OF=1，若符号位没有变化，则认为没有溢出，OF=0。

例 5. 3. 16 （AL）=11100010B，将（AL）算术左移 1 位。

```
SAL  AL，1
```

移位后（AL）为 1 1 0 0 0 1 0 0 ，（AL）移位前的最高位移入 CF 中，即 CF=1，最低位被补充 0,并且由于移位前后符号未变，补码自加没有溢出，OF=0。

例 5. 3. 17 （AL）=11100010B，将（AL）算术左移 2 位。

```
MOV  CL，2
SAL  AL，CL
```

移位前（AL）为 1 1 1 0 0 0 1 0
移位后（AL）为 1 0 0 0 1 0 0 0 ，（AL）移位前的第 6 位移入 CF 中，即 CF=1，最低两位被补充 0，由于移位位数大于 1，OF 标志无效。

7） 算术右移指令

指令格式：SAR DEST，COUNT

COUNT 的含义为移位位数，解释与 SAL 指令相同。

指令功能：将（DEST）右移 COUNT 位，从右边移出的最高一位保存到 CF 中，左边空出的数据位则补充（DEST）的符号位，其含义为补码除以 2^{COUNT}。

标志位影响：所有状态标志，包括 OF、SF、ZF、AF、PF、CF，其中 AF 标志不确定，其余标志位的解释与算术运算指令一致，OF 在算术右移中始终为 0，参见例中解释。

与 SAL 指令的功能相反，SAR 指令的功能是将（DEST）解释为补码，并将其除以 2 的 COUNT 次方。移位后在左边空出的数据位补充符号位是为了保持补码的性质不被改变。与 SAL 指令类似，SAR 指令对操作数的解释没有二意性，固定将（DEST）解释为补码。

SAL SAR 都将目的操作数看作补码

例 5. 3. 18 （AL）=11100010B，将（AL）算术右移 2 位。

```
MOV  CL，2
SAR  AL，CL
```

由于右移操作使补码真值的绝对值越来越小，因此在算术右移中不可能出现溢出，OF=0

8） 逻辑左移指令

指令格式：SHL DEST，COUNT

COUNT 的含义为移位位数，解释与 SAL 指令相同。

标志位影响：所有状态标志，包括 OF、SF、ZF、AF、PF、CF，其中 AF 标志不确定，其余标志位的解释与算术运算指令一致，但仅当 COUNT=1 时，OF 标志才有意义。

SHL 指令的功能与 SAL 指令相同，都是完成（DEST）的左移操作，只是 SHL 指令将操作数解释为无符号数编码。

9） 逻辑右移指令

指令格式：SHR DEST，COUNT

COUNT 的含义为移位位数，解释与 SAL 指令相同。

标志位影响：所有状态标志，包括 OF、SF、ZF、AF、PF、CF，其中 AF 标志不确定，其余标志位的解释与算术运算指令一致，但仅当 COUNT=1 时，OF 标志才有意义。

SHR 指令的功能与 SHL 指令相反，它将（DEST）解释为无符号数编码，应注意 SHR 指令与 SAR 指令的区别，它们是不同的指令，SAR 指令在左侧补充符号位，而 SHR 指令在左侧补充 0

逻辑右移指令仍然按照算术右移指令的规则来判断 OF 标志，但由于其操作数解释为无符号数编码，因此，OF 标志在多数情况下没有应用价值，只能用于体现最高位在移位前后有无变化，而不能作为符号变化来解释

判断 OF 标志：看符号位是否发生变

化（仅仅针对补码，也就是有符号数）

算术左/右移：补码

逻辑左/右移：原码

（10）SHR BL，3

错误，当移位次数大于 1 时，在移位指令中特定使用 CL 寄存器给出移位次数

正确的写法：MOV CL,3

SHR BL,CL

例 5. 3. 19 （AL）=11100010B，将（AL）逻辑右移 2 位。

```
MOV  CL，2
SHR  AL，CL
```

10） 循环左移指令

指令格式：ROL DEST，COUNT

指令功能：将（DEST）左移 COUNT 位，从左边移出的所有数据位按原有顺序补充到右边，从左边移出的最低一位保存到 CF 中。

标志位影响：CF、OF。CF 用于保存从左边移出的最低一位，OF 的判断规则与算术左移指令一致，当 COUNT=1 时，若移位前后（DEST）的最高位未发生变化，则 OF=0，若发生变化，OF=1；当 COUNT>1 时，OF 无意义。

例 5. 3. 20 （AL）=11100010B，将（AL）循环左移 2 位。

```
MOV  CL，2
ROL  AL，CL
```

移位前（AL）为 1 1 1 0 0 0 1 0

移位后（AL）为 1 0 0 0 1 0 1 1，（AL）移位前的第 6 位被移入 CF，即 CI

由于移位位数大于 1，因此 OF 无意义。

11） 循环右移指令

指令格式：ROR DEST，COUNT

COUNT 的含义为移位位数，解释与 SAL 指令相同。

指令功能：将（DEST）右移 COUNT 位，从右边移出的所有数据位按原有顺序补充到左边，从右边移出的最高一位保存到 CF 中。

标志位影响：CF、OF。CF 用于保存从左边移出的最低一位，OF 的判断规则与算术左移

令一致，当 COUNT=1 时，若移位前后（DEST）的最高位未发生变化，则 OF=0，若发

变化，OF=1；当 COUNT>1 时，OF 无意义。

12） 带进位循环左移指令

指令格式：RCL DEST，COUNT

COUNT 的含义为移位位数，解释与 SAL 指令相同。

指令功能：将（CF）看作（DEST）的最高位，将（CF）、（DEST）构成的整体数据循环左移 COUNT 位。

标志位影响：CF、OF。CF 作为移位数据的最高位，OF 的判断规则与算术左移指令一致，当 COUNT=1 时，若移位前后（DEST）的最高位（指存储单元最高位，不是 CF）未发生变化，则 OF=0，若发生变化，OF=1；当 COUNT>1 时，OF 无意义。

划重点：

例 5. 3. 21 假设一个 48 位的无符号数编码由低位到高位分别存放于 DA、DA+2、DA+4 这三个字内存单元（16 位）中。要求实现此 48 位无符号数编码乘以 2 的运算。

乘以 2 的运算即为左移 1 位，程序片段如下：

```
SHL  DA，1
RCL  DA+2，1
RCL  DA+4，1
```

每次低地址字的最高位都移位至 CF 保存，下次移位时，CF 被移位至高地址字的最低位，而高地址字的最高位又被移位至 CF 保存，如此继续，直到所有存储单元处理完毕。

长操作数左移操作是由低位到高位的顺序进行的，并且第一条指令使用 SHL（SAL）指令，它是常规的左移指令，后续移位操作才使用 RCL 指令。

无符号数编码自加溢出判断使用 CF 标志，补码自加溢出判断使用 OF 标志。

并且，无论是无符号数编码还是补码，只有当最高字（字节）完成左移操作后，即对长操作数的移位完全完成后，才能对移位是否溢出进行判断。

对于长操作数的移位操作只能逐位进行，不能一次移动多位。原因有两个，第一，CF 每次只能容纳一个被移出的位，第二，同时移动多位将导致 OF 标志无意义，无法对补码移位的溢出进行判断.

13） 带进位循环右移指令

指令格式：RCR　DEST，COUNT

COUNT 的含义为移位位数，解释与 SAL 指令相同。

标志位影响：CF、OF。CF 作为移位数据的最高位，OF 的判断规则与算术左移指令一致，当 COUNT=1 时，若移位前后（DEST）的最高位（指存储单元最高位，不是 CF）未发生变化，则 OF=0，若发生变化，OF=1；当 COUNT>1 时，OF 无意义。

例 5. 3. 22 假设一个 48 位的无符号数编码由低位到高位分别存放于 DA、DA+2、I
这三个字内存单元（16 位）中。要求实现此 48 位无符号数编码除以 2 的运算。

除以 2 的运算即为右移 1 位，程序片段如下：

```
SHR　DA+4，1
RCR　DA+2，1
RCR　DA，1
```

长无符号数编码右移时第一条指令使用 SHR 指令，长补码右移时第一条指令使用 SAR 指令，与左移操作不同，SHR 与 SAR 是不同的指令，因此在程序设计中一定要明确所使用的编码是无符号数还是补码。

SHL（SAL）与 RCL 指令配合可实现长操作数左移操作，操作顺序是由低位到高位，SHR 或 SAR 指令与 RCR 指令配合可实现长操作数右移操作，操作顺序是由高位到低位。

4. 处理器控制类指令

1）进位标志清除指令

指令格式：CLC

指令功能：清除 CF 标志，使 CF=0

标志位影响：CF

该指令最容易理解的用途为配合 ADC 指令实现长操作数加法。进入程序的循环结构前将 CF 清零，循环结构中仅使用 ADC 指令完成长操作数各字（字节）的带进位加法，但第一次执行循环时，由于 CF=0，则 ADC 指令等同于 ADD 指令，如此可省略对 ADD 指令的使用，简化程序结构。

2）进位标志置位指令

指令格式：STC

指令功能：置位 CF 标志，使 CF=1

标志位影响：CF

3）进位标志取反指令

指令格式：CMC

指令功能：将 CF 标志取反

标志位影响：CF

4）方向标志清除指令

指令格式：CLD

指令功能：清除 DF 标志，使 DF=0

标志位影响：DF

5）方向标志置位指令

指令格式：STD

指令功能：置位 DF 标志，使 DF=1

标志位影响：DF

6）中断使能标志清除指令

指令格式：CLI

指令功能：清除 IF 标志，使 IF=0

标志位影响：IF

关闭 CPU 内部的可屏蔽中断响应开关，当设备接口向 CPU 提出可屏蔽中断请求时，CPU 不会响应。

7）中断使能标志置位指令

指令格式：STI

指令功能：置位 IF 标志，使 IF=1

标志位影响：IF

开启 CPU 内部的可屏蔽中断响应开关，当设备接口向 CPU 提出可屏蔽中断请求时，CPU 会做出响应。

8）停机指令

指令格式：HLT

指令功能：切断 CPU 的时钟输入，系统停机

标志位影响：无

该指令执行后，CPU 的时钟信号输入被切断，CPU 无法继续执行所有时序过程，处于“休克”状态，系

统停机。执行 HLT 指令后，仅能通过 RESET 复位信号使系统重新启动， 才能恢复 CPU 的运转。

9）空操作指令

指令格式：NOP

指令功能：无操作，CPU 空转 3 个节拍

标志位影响：无

在调试可执行程序时，如果调试人员想要屏蔽某些机器指令功能，那么使用 NOP 指令最合适，因为它是单字节的机器指令，任意长度的机器指令都可以由它来填充并屏蔽，这是 NOP 指令最大的应用价值。除此之外，有时也可以使用多条 NOP 指令来进行延时。当然，NOP 指令的延时功能具有较差的移植性，如果读者需要精确延时，建议使用定时器中断相关的系统调用。

本章练习：

4. 按要求分析下面程序片段的执行结果。

MOV AL，0C2H

MOV AH，0E4H

ADD AL，AH

执行该程序片段后，（AL）=？，（AH）=？，如果将 ADD 指令的两个操作数解释为无符号数，运算有没有溢出？为什么？如果将 ADD 指令的两个操作数解释为补码，运算有没有溢出？为什么？

答：执行该程序片段后，（AL）=0A6H，（AH）=0E4H，如果操作数解释为无符号数，运算溢出，因为加法运算后最高位产生了进位，CF=1，需使用 9 个二进制位才能表达完整运算结果；如果将操作数解释为补码，则运算没有溢出，因为从操作数与运算结果的符号位观察，两个操作数均为负数补码，相加后所得结果仍然为负数补码，符号位正确，表明加法结果未超出补码表示范围（这里是 8 位补码的表示范围），加法运算后 OF=0。

看 OF：把需要操作的两个数看作补码，再看结果符号位是否有变化

5. 按要求分析下面程序片段的执行结果。

MOV AL，98H

MOV BL，42H

XCHG AL，BL

SUB AL，BL

执行该程序片段后，（AL）=？，（BL）=？，如果将 SUB 指令的两个操作数解释为无符号数，运算有没有溢出？为什么？如果将 SUB 指令的两个操作数解释为补码，运算有没有溢出？为什么？如果将 SUB 指令的两个操作数解释为补码，其减法运算对应的十进制真值表达式应如何书写？

Al：10011000

Bl：01000010

Bl：10011000 al：01000010

Al=01000010-

10011000=01000010+01101000

=10101010

答：执行该程序片段后，（AL）=0AAH，（BL）=98H，如果将操作数解释为无符号数，则运算溢出，从操作数判断，此运算属于被减数小于减数的情况，这在无符号数运算中是不允许的（如果当前操作数仅为长数据的一部分，则另当别论），减法运算后最高位必然产生借位，CF=1；如果操作数解释为补码，运算也溢出，从操作数判断，此运算属于“正-负”类型，等价于“正+正”类型，正确的运算结果应为正数或零的补码，而运算结果的符号位却为“负”，表明运算结果超出补码表示范围（这里为 8 位补码表示范围），减法运算后 OF=1。

SUB 指令所使用的被减数补码为 42H=01000010B，减数补码为 98H=10011000B，由于被减数为正数补码，它等于真值本身，而减数补码为负数补码，将其取反加 1 后，添上负号，得到其二进制真值为 -01101000B。将被减数、减数的二进制真值转换为十进制后，得到真值运算表达式：66 -（-104）=170，很明显运算结果超出 8 位补码的最大值+127。

6. 按要求分析下面程序片段的执行结果。

STC

MOV AL，03H

AND AL，02H

ADC AL，00H

执行该程序片段后，（AL）=？

Al： 00000011

00000010 ->00000010

注意：and 指令强置 CF=0，AF 不确定

7. 假设（DS）=1000H，（SS）=2000H，字内存单元（10200H）=0870H，（10202H）=2000H，（20870H）=0203H，（20872H）=0405H，括号内所给为内存单元物理地址，括号表示该地址所指示单元中保存的数据，分别执行下列程序片段后，按要求分析各程序片段的执行结果。

（1）MOV AL，[0200H]

执行该程序片段后，（AL）=？

答：源操作数地址为（DS）*16+0200H=10000H+0200H=10200H，因此执行该程序片段后，（AL）=70H （逆序存放，低地址对应低数据位）

（2）MOV BP，0871H

MOV BL，[BP]

执行该程序片段后，（BL）=？

答：第二条指令的源操作数地址为（SS）*16+（BP）=20871H，执行该程序片段后，（BL）=02H （逆序存放，高地址对应高数据位）

（3）LEA SI，[0200H]

执行该程序片段后，（SI）=？

答：LEA 指令将源操作数的 EA 传送到目的操作数保存，（SI）=0200H

LEA：地址 MOV：数据

8. 按要求分析下面程序片段的执行结果。

MOV AX，651CH

SHL AL，1

RCL AH，1

执行该程序片段后，（AX）=？，该程序片段的功能是什么？如果将（AX）解释为无符号数，那么运算是否溢出？为什么？如果将（AX）解释为补码，运算是否溢出？为什么？SHL 与 SAL 指令间有什么关联和区别？

SHL：逻辑左移（有符号数

SAL：算术左移

Ax： 01100101 00011100

11001010 00111000 ->CA38H

答：执行该程序片段后，（AX）=0CA38H，该程序片段的功能为将 AX 中的 16 位编码左移 1 位，等价于乘以 2（也可理解为自加一次）。如果将（AX）解释为无符号数，那么运算没有溢出，因为最后一次移位操作后，最高移出位为 0，即 CF=0（自加完成后最高位无进位）；如果将（AX）解释为补码，运算溢出，因为移位前后（AX）的最高位发生了变化（由 0 变为 1），符号位在运算中丢失，可以理解为自加运算结果超出了 16 位补码表示范围。

9. 按要求分析下面程序片段的执行结果。

MOV AL，35H

AND AL，0FH

执行该程序片段后，（AL）=？CF、OF、AF、ZF、SF、PF 标志取值是什么？该程序片段的功能是什么？

00110101

00001111 ->00000101

AL=05H

CF=0 OF=0 AF 不确定 ZF=0 SF=0 PF=1

10. 假设一个 48 位的补码按照由低位到高位的顺序保存在字类型的内存单元 VA1、VA1+2、VA1+4 中，试按下列要求完成程序片段设计。(**红字部分请在教材中纠正**)

（1）设计程序片段，实现将该 48 位补码除以 4 的功能，运算结果仍然保存在原内存单元中。

```
SAR VA1+4,1
RCR VA1+2,1
RCR VA1,1
```

（2）设计程序片段，求该 48 位补码的相反数补码，运算结果仍然保存在原内存单元中。

```
NOT VA1
NOT VA1+2
NOT VA1+4
ADD VA1,1
ADC VA1+2,0
ADC VA1+4,0
```

12. 假设（SP）=0060H，执行两次 PUSH 指令后，（SP）=？假设（SP）=0038H，执行三次 POP 指令后，（SP）=？

Push 一次减 2 pop 一次+2

13. 按要求分析下面程序片段的执行结果。

MOV AL，01H

NEG AL

INC AL

执行该程序片段后（AL）=？，CF、OF 标志的状态是什么？

AL:00000001

NEG:11111111(取相反数补码)

INC:00000000（加一）

答：执行该程序片段后（AL）=0，CF=1，注意，这是受 NEG 指令影响的结果，INC 指令不影响 CF 标志；OF=0，加法运算并无溢出，因为 0FFH 为-1 的补码，加 1 后等于 0 是正确的。

求补指令 NEG (negate)
指令对标志位的影响：

CF=1 不为0的操作数求补时
CF=0 为0的操作数求补时
OF=1 操作数为-128（字节运算）或操作数为-32768（字运算）
OF=0 当求补运算的操作数不为－128（字节）或－32768（字）时

14. 按要求分析下面程序片段的执行结果。

MOV BL，51H

AND BL，0FEH

XOR BL，50H

DEC BL

执行该程序片段后（BL）=？，CF、OF 标志的状态是什么？

BL:01010001

11111110 ->01010000

01010000 ->00000000

DEC：自减 11111111->0FFH

答：执行该程序片段后（BL）=0FFH，CF=0，注意，这是受 XOR 指令的影响，XOR 指令将 CF 强置为 0，而 DEC 指令不影响 CF；OF=0，此标志是受 DEC 指令影响的结果。此题中应注意，逻辑运算指令会将 CF、OF 强置为 0，而 DEC 指令不影响 CF 标志。

15. 按照各小题的要求分别设计程序片段。

（因存在多种设计方式，程序设计题目的答案仅作为参考）

（1）将 AL 寄存器的高 4 位与低 4 位交换

```
MOV CL,4
ROL AL,CL(循环左移)
```

（2）将 TF 标志位置 1

```
PUSHF
POP  AX
OR   AX,0100H
PUSH AX
POPF
```

（3）将 AL 寄存器的第 7 位清 0，但不影响其它数据位

```
AND AL,07FH
```

（4）分离 AL 寄存器的最低两位，其它数据位清 0。

```
AND AL,03H
```

（5）分离 AL 寄存器的高 4 位与低 4 位，并分别保存在 BL、BH 的低 4 位

```
PUSH AX(临时保存数据)
AND AL,0FH 0000FFFF 只有后四位
MOV BL,AL
POP AX
MOV CL,4
SAR AL,CL(右移 4 位)
MOV BH,AL
```

第六章 汇编语言源程序组织

1. 语句种类和格式

1）指令语句

一般格式：

标号： 指令助记符 目标操作数表达式，源操作数表达式 ；注释

```
Next:   MOV   al,[si]   ;取出一个字节的内容送 al
```

其中，标号是指令语句的符号地址。其实源程序经编译（汇编）后，每条指令语句的代码都会按先后顺序自然包含程序运行时在代码段的存放位置，即每条指令语句都有各自的地址，而标号是给要使用这条语句地址的代码作了一个标记，程序编译（汇编）时会将其他使用到这个符号地址的语句中的这个标号替换成相应的地址。比如在同一代码段内其他位置有一条跳转指令：

```
Jmp     Next
```

无论一条指令语句是否有标号，编译后的代码是没有区别的。

2）伪指令语句

伪指令是用于指示编译（汇编）程序如何编译（汇编）源程序的，例如描述源程序如何分段和各段分别由哪个段寄存器指向，以及常量和变量及子程序的定义等

伪指令是用于控制源程序编译（汇编）而不是控制 CPU 运行的；伪指令是在源程序编译时被执行的，而指令是在程序运行时被执行的

下面是伪指令语句的一般格式：

```
符号名  伪指令  操作数表达式 1,操作数表达式 2,...,操作数表达式 n    ;注释
```

和指令语句在形式上的区别除语句核心字段是伪指令而不是指令外，符号名后没有冒号也是明显和重要的区别特征。

语句中的“符号名”是常量、变量、子程序或段结构等的命名。

3）标识符

I. 标识符是长度不超过 31 的字符串

II. 标识符的字符集是英文字母、数字和下述特殊字符：

? @ _ . $

的 ASCII 码，但第一个字符不能是数字

III. 标识符不能和汇编语言的保留字冲突。即不能和寄存器、指令、伪指令等同名。

2. 常量与变量

1）常量

I. 数字常量

① 十进制数：以字母 D 结尾（也可以忽略）的 0～9 组成的数字序列。

② 二进制数：以字母 B 结尾的 0 和 1 组成的数字序列。如 01001101B。

③ 十六进制数：以字母 H 结尾的 0～9 及 A～F 组成的序列（这里 A～F 对应 10 进制数 10～15）；当十六进制数的高位是 A～F 的某个字母时，为了避免与标识符混淆，规定在这种情况下十六进制数前面必须加一“0”作为区别。如：0F4H。

④ 八进制数：以字母 O 结尾的 0～7 的数字序列。

⑤ 浮点数。如：12.34E-6（表示 12.34×10^{-6}）。

II. 字符串常量

字符串常量由单引号或双引号括起来的 1 个或多个字符组成，如：

’Hello, World!’

’G’

编译（汇编）后每个字符被翻译成对应的 ASCII 码，每个字符占一个字节。如’ABC’编译（汇编）后所占 3 个字节的内容是 41H、42H、43H。

```
        ADD   AL,6          ;用作立即数
        MOV   AH,[100H]     ;用作直接寻址方式中的偏移量
        MOV   AL,[BX+20]    ;用作基址或变址寻址方式中的位移量
X       DB    20H           ;给变量 X 赋初值
```

III. 符号常量定义伪指令

① 等值伪指令 equ

```
Pi      equ     3.1415926
```

这条语句给编译（汇编）程序的指示是：将源程序中所有的 Pi 都用 3.1415926 替代。

② 等号伪指令“=”

简单地将“equ”改写成“=”：

```
Pi      =       3.1415926
```

“=”伪指令和“equ”的差别是：后者可以对同一常量作多次定义，比如在程序的另一处可以：

```
Pi      =       3.1416
```

即此后遇到 Pi 的地方都被编译成 3.1416 而不是 3.1415926。需要说明的是，尽管这条伪指令有点灵活性，但笔者认为这样做会增加编程时为避免混乱的额外负担，所以建议定义常量就使用 equ 伪指令即可。

2）简单变量定义

计算机存储单元的内容是可以被读取或被修改的，所以编程语言用符号命名的若干存储单元来表示一个变量。汇编语言中，常见的数据类型有字节、字和双字等。

I. 变量定义的一般形式

```
[变量名]    数据定义符    表达式 1[, 表达式 2, ..., 表达式 n]    [;注释]
```

① 变量名必须是一个合法的标识符；

② 数据定义符用于确定变量的数据类型，常用的定义符有：DB、DW 和 DD 等；

③ 表达式用于定义内存单元的初值，一个定义语句可以有多个表达式，各表达式之间必须用西文逗号‘,’分开；如果某个存储单元没有初值表达式，则必须用一个西文问号‘?’来替代；

④ 变量定义语句的注释根据具体需要，可写可不写。

其实变量名就是某个或某若干个字节存储单元的符号地址，这和指令语句的标号很相似：变量名是所描述的变量在数据段的偏移量，这是汇编程序在编译（汇编）时，将操作数第一个字节的偏移地址赋给了这个符号。

II. 定义字节变量

字节变量的定义符为 DB，如

```
X       DB      0
```

该语句定义 X 为字节型变量，分配给 X 一个字节，且初值为零。

这条伪指令语句和下面 C 语言的变量定义语句一样的

```
char        x = 0;
```

```
X       DB      ?
```

这里的西文问号表示没有给 X 赋初值，但分配一个字节的空间。这和 C 语言的变量定义语句是一样的：

```
char        x;
```

Eg :

```
ODD_TABLE DB  1,3,5,7,9
HEX_TABLE DB '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E', 'F'
```

前一语句为字节型变量 ODD_TABLE 分配了 5 个字节并依序被初始化成了 10 以内的奇数；后一语句定义了一个 16 进制数的 ASCII 码表。

DB 伪指令还支持用“字符串”表达式定义变量初值，如

```
MSGHI       DB      'Hello,World!'
```

其实，前面提到的“16 进制数的 ASCII 码表”也可以这样定义：

```
HEX_TABLE DB '0123456789ABCDEF'
```

这样显然简洁得多。

III. 字变量定义

字变量的定义符为 DW；每个字占用两个连续的字节单元。如：

```
WVAR    DW    2008H
```

上述定义初始化后的内存分配如下所示：

注意字变量数据是按“高位存放到高地址字节、低位存放在低地址字节”的方式存于存储单元之中的，而字节数据是按照顺序排列在存储单元中的

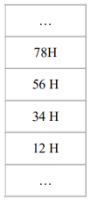
```
BVAR    DB      20H,08H
```

数据在内存里是这样存放的：

IV. 双字变量

双字变量的定义符为 DD，每个双字变量占用二个连续的字单元共四个字节。

```
DDVAR       DD      12345678H
```



字和双字类型变量多用作存储较大值域的数据，不适合用作存储 8 位字符的字符串。

V. 重复说明符 DUP

我们知道 C 语言是这样定义数组的：

```
char x[100];
```

而汇编语言用“重复说明符”DUP 来定义同类型变量：

```
BVAR    100    DUP    (?)
```

带重复说明符 DUP 的表大式的一般形式是：

```
count  DUP  (表达式, 表达式, …, 表达式)
```

其中：count 是重复次数，括号内的若干表达式是被重复的部分；“表达式”可以直接是存储单元的初值，也可以是另一个被嵌套的 DUP 表达式；表达式括号中的多个表达式用西文逗号“,”分开。下面是一些使用 DUP 的例子：

```
WVAR    DW    50    DUP (?)          ;定义了 50 个字单元变量
STRHI   DB    20    DUP ('Good!')    ;定义了 100 个字节的字节型变量并在这
                                     ;100 个字节的存放了 20 个'Good!'。
```

3）标号和内存变量的属性及属性操作符

I. 段属性运算符 SEG

SEG 指示编译程序返回变量或标号操作数所在段的段基址。

```
BVAR    DB      200 DUP(?)
WVAR    DW      100 DUP(0)

MOV     SI,SEG BVAR
MOV     DI,SEG WVAR
```

因为 BVAR 和 WVAR 在同一数据段中，所以编译后，SEG BVAR 和 SEG WVAR 是相同的，且和变量本身的类型无关。

II. 偏移量属性运算符 OFFSET

偏移量属性运算符指示编译程序返回变量或标号操作数

的偏移量。

```
WVAR    DW      100 DUP(0)
```

而下述代码：

```
MOV     SI,OFFSET WVAR
```

的源操作数表达式在程序编译时将得出一个立即数的结果；程序执行这条语句时，SI 将获得变量 WVAR 的偏移量

和 LEA 指令相比，指令执行后的效果是一样的，但实现的手段是有区别的：LEA 是纯指令实现的功能，而偏移量运算符主要是靠编译程序的功能实现的；另外 LEA 的目标操作数只能是 16 位通用寄存器，而带 OFFSET 运算符及变量的表达式经编译后是一个立即数，所以目标操作数还可以是 16 位存储器单元。

III. 类型属性运算符 TYPE

类型属性运算符 TYPE 返回变量的字节数或语句标号的 FAR 或 NEAR 类型

标号及变量类型属性表

	类型属性	类型数字
变量	BYTE	1
	WORD	2
	DWORD	4
标号	NEAR	-1
	FAR	-2

变量的类型数字就是变量的字节数，直观实用；而标号的属性用于说明该标号能否被其他段的代码引用

NEAR	表示该标号只能被同段代码引用；
FAR	表示该标号能被其他段代码引用。

语句标号的属性在语句定义时，被默认成 NEAR 属性；

标号的 FAR 属性的须通过 LABEL 伪指令设置。

IV. LABEL THIS PTR

LABEL 伪指令用于设置或更改变量或标号的属性：

```
变量名    LABEL     类型

标号      LABEL     类型
```

其中变量的数据类型可以是 BYTE，WORD，DWORD；标号的类型可以是 NEAR 或 FAR。

```
WVAR    LABEL   WORD
BVAR    DB      100 DUP(0)
```

这样定义后，就可以用字节或字属性访问这 100 个字节了。比如在做数据块清零或复制等操作时，用字属性操作是 16 位的，代码效率会更高。

```
NEXTF   LABEL   FAR
NEXT:   MOV     SI, BX
```

为上述数值语句定义了不同属性的标号，段内转移指令使用标号 NEXT，段间使用 NEXTF

目标代码和没有标号说明时占用的空间是一样的，只是编译（汇编）程序会将同段其他位置的标号替换成该语句的段内偏移量

在上述语句中使用 EQU THIS 替换 LABEL 可以实现同样的目的

```
WVAR    EQU THIS    WORD
BVAR    DB          100 DUP(0)

NEXTF   EQU THIS    FAR
NEXT:   MOV         SI, BX
```

如果只是个别地方需要用和定义时不同的属性对该变量

进行访问，可使用强制类型转换运算符 PTR 来实现。

```
BVAR    DB          100 DUP(0)
```

而这样访问它：

```
MOV     AX, BVAR
```

编译（汇编）会因类型不匹配而不能通过。而将语句改成：

```
MOV     AX, WORD PTR BVAR
```

就没有问题了。强制类型转换字段的一般格式为：

```
数据类型  PTR  地址表达式
```

强制类型转换给变量赋予一种临时性的属性，只在本条语句有效

V. LENGTH 和 SIZE

长度属性操作符 LENGTH 是内存变量操作符，它指示编译（汇编）程序返回重复操作符 DUP 中的重复数。如果有嵌套的 DUP，则只返回最外层的重复数；如果 DUP，则返回 1。显然，这个操作符的功能是很有限的，完全不能和 C 语言的 Strlen 函数相提并论。

有同样毛病的是容量属性操作符 SIZE，它的返回值按下列公式计算：

SIZE 变量 =（LENGTH 变量）×（TYPE 变量）

3. 汇编语言表达式

高级语言涉及到变量的运算表达式将用于在目标代码运行时控制计算机的运作，而汇编语言的表达式是用于控制编译（汇编）的

```
S=3.1415926*r*r;
```

如果 r 是变量，那么表达式 3.1415926*r*r 就是用于控制计算机运作的代码，其结果是在该语句执行后确定的；而汇编语言的表达式在编译（汇编）后已经是被确定的常量了。

1） 数值表达式

数值表达式是在编译（汇编）过程中能够确定计算值的表达式，即编译（汇编）完成后表达式的值就完全确定

I. 算术运算符

符号：+（正）、-（负）；

运算符：+（加）、-（减）、*（乘）、/（除）和取模 MOD。

II. 关系运算符

关系运算符包括符号：EQ（相等）、NE（不等）、LT（小于）、GT（大于）、LE（小于等于）和 GE（大于等于）。

若关系不成立，数值表达式的结果为 0，否则为 0FFFFH。

```
1+1  GT 2          ;结果为 0（假）
1+1  EQ 2          ;结果为 0FFFFH（真）
```

III. 逻辑运算符

AND（逻辑与）、OR（逻辑或）、NOT（逻辑非）、XOR（异或）、SHL（左移位）和 SHR（右移位）

```
1 SHL 3，47H AND 0FH，NOT 56H
```

它们的计算结果分别为：8，7 和 0A9H。

IV. HIGH 和 LOW 操作符

HIGH 和 LOW 操作符分别获取表达式计算结果的高 8 位和低 8 位

```
HIGH    表达式
LOW     表达式
如：
HIGH    8A00H+126H      ;表达式的结果为 8BH
LOW     8A00H+126H      ;表达式的结果为 26H
```

V. 运算符和操作符的优先级

汇编语言运算符和操作符的优先级

优先级：高	LENGTH、SIZE
	PTR、SEG、OFFSET、TYPE、THIS、:（用于段超越前缀）
	*、/、MOD、SHL、SHR
	HIGH、LOW
	+、-
	EQ、NE、LT、LE、GT、GE
	NOT
	AND
优先级：低	OR、XOR

2） 地址表达式

地址表达式是计算存储单元地址的表达式，它可由标号、变量名和由括号括起来的基址或变址寄存器组成。其计算结果表示一个存储单元的地址，而不是该存储单元的值。

```
BVAR    DB      1, 2, 3
WVAR    DW      1234H, 5678H
```

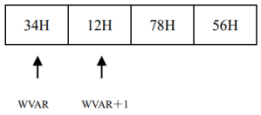
则语句：

```
MOV     AL, BVAR+1
```

执行后，AL 的内容为 2，即编译（汇编）后，地址表达式 BVAR+1 指向变量 BVAR 之后内容为 2 的那个字节

```
MOV     AX, WVAR+1
```

执行后，AX 的内容不是 5678H，而是 7812H。这是因为地址表达式 WVAR+1 没有指向变量 WVAR 之后的那个字而是之后的那个字节




```
BVAR    DB      ?, ?

VOFF    DW      BVAR+1
VPTR    DD      BVAR+1
```

似乎定义 VOFF 和 VPTR 的表达式包含了变量，事实上 BVAR 是一个符号地址，经编译（汇编）后已经是一个确定的常量，故表达式 BVAR+1 也是一个常量：变量 BVAR 第二个字节的偏移量，作为变量 VOFF 的初值存放到偏移量为 VOFF 的字单元中了；而当双字变量 VPTR 的初值为包含变量的表达式时，编译（汇编）程序将地址表达式指向的变量的 32 位地址指针赋给 VPTR。

4. 段定义伪指令与源程序框架

```
stack0  segment stack       ;堆栈段
        dw 40h dup(0)
stack0  ends

data    segment             ;数据段
Hello   db "Hello,World!",0
data    ends

code    segment             ;代码段
        assume cs:code,ds:data,ss:stack0
main:   mov ax,data         ;ds指向数据段
        mov ds,ax
        mov ax,0B800h
        mov es,ax

        mov dh,10
        mov dl,8
        mov ah,1eh
        lea si,Hello
        call Direct         ;调用子程序

        mov ah,0
        int 16h

        mov ah,4ch          ;程序结束，退回
        int 21h             ;DOS

Direct  proc        ;显示子程序
        push bx
        push di
        mov bh,ah
        mov al,dh
        dec al
        mov bl,160
        mul bl
        mov di,ax
        dec al
        mov al,dl
        mov bl,2
        mul bl
        add di,ax
Next:   mov al,[si]
        cmp al,0
        jz  return
        mov es:[di],al
        inc di
        mov es:[di],bh
        inc di
        inc si
        jmp Next
return: pop di
        pop bx
        ret
Direct  endp
code    ends
        end main
```

1） 段定义伪指令

SEGMENT 和 ENDS，这是汇编语言段定义伪指令。

一般格式：

```
段名    SEGMENT  [对齐类型] [组合类型] [类别]
        [段内语句序列]

段名    ENDS
```

段名是由编程者根据段的用途按标识符的规则定义的，且前后二个段名要相同。和其他写在一行的伪指令语句不同，段定义伪指令和结构定义语句相似，或者也可以看作是一种语句括号：以关键字 SEGMENT 开头，以段结束伪指令 ENDS 结束。

类似的这种伪指令语句还有过程定义语句和宏定义语句等。和其他伪指令语句定义的符号名的另一个不同是，汇编语言程序的段名可以是唯一的，也可以与其它段同名。如果有二个段同名，则后者被认为是前段的后续，它们属同一段。

关键字 SEGMENT 之后的“对齐类型”、“组合类型”和“类别”为段属性。段属性必须按如上所述的顺序说明，而是否描述可根据需要选择

对齐类型

对齐类型包括 BYTE、WORD、DWORD、PARA 和 PAGE 共 5 个类型，用于描述当前段对起始地址的要求：

¬ BYTE：逻辑段从未占用的下一个字节开始。

¬ WORD ： 逻辑段从未占用的下一个字开始。有可能空闲 1 个字节；段的起始地址必须是偶数。

¬ DWORD ： 逻辑段从未占用的下一个双字开始。有可能空闲 3 个字节；段起始物理地址的低 2 位为 0。 ¬ PARA：逻辑段从

下一个未占用的节（16 个字节）开始，有可能空闲 15 个字节；段起始物理地址的低 4 位为 0。当未指定对齐类型时，节对齐为默认对齐类型。

¬ PAGE：逻辑段从下一个未占用的页（256 字节）开始。有可能空闲 255 个字节， 段的起始物理地址的低 8 位为 0

组合类型

组合类型是告诉连接程序如何把不同模块中段名相同的段合并在一起。具体的组合类型如下：

¬ NONE：表示当前段在逻辑上独立于其它模块，并有其自己的基地址。NONE 是缺省的组合类型。

¬ PUBLIC：表示当前段与其它模块中同段名的 PUBLIC 类型段组合成一个段。组合的先后次序取决于 LINK 程序中目标模块排列的次序。在组合时，后续段的起始地址要按其对齐类型进行定位，所以，同名段之间可能有间隔。

¬ COMMON：表示当前段与其它模块中同名段重叠，也就是说，它们的起始地址相同。最终段的长度是同名段的最大长度。由于段覆盖，所以，前一同名段中的初始化数据被后续段的初始数据覆盖掉。

¬ STACK：组合类型 STACK 表示当前段是堆栈段，其组合情况与 PUBLIC 相同。

¬ AT 数值表达式：以数值表达式的值作为前段所的起始地址。 例子中除了堆栈段的组合类型是 STACK 外，其他的段都没有明确描述组合类型，即为默认的 NONE 独立组合类型。

类别

类别是一个由程序员按标识符规则设定的带单引号的字符串；连接程序将类别名相同的段组合起来在存储器中连续存放。

2） 段声明伪指令的段初值

```
assume   cs:code,ds:data,ss:stack0
```

这条语句通过建立了段名和段寄存器间的对应关系，让编译（汇编）程序知道了程序的段是如何安排的。ASSUME 语句的一般形式是：

```
assume  段寄存器名:段名[, 段寄存器名:段名, ……]
```

虽然 assume 伪指令指明了某段名和某段寄存器的关系，但并没有确定段寄存器的初值。

当典型的 DOS 程序运行时，操作系统会根据内存的使用情况把执行程序引导、安放到内存的适当位置，这里所说的“安放”就是通过初始化段寄存器来实现的。这样做不仅让我们在编写单个程序时不用去考虑段间冲突，也不用担心两个执行程序在内存出现冲突。DOS 在引导程序进入内存时对各个段寄存器的初始化是有区别的：CS 和 SS（当 对应段的组合类型定义成 STACK 时）被直接初始化了，而在装入程序过程中 DS 和 ES 寄存器有其它用途，所以只能在用户程序中用指令专门对其作初始化，以装入用户数据段和附加段的基址。

```
mov     ax,data
mov     ds,ax
```

通过 ax 搭桥的原因是由于所有的段寄存器都不能接收立即数；而 data 和其他段名一样是一种特殊的立即数：它们的性质一直是立即数，但它们的值是在程序装入时由操作系统确定的。

3） IP 和 SP 的初值

我们先说说给堆栈指针 SP 赋初值。如果对应段的组合类型选择 stack，栈指针 SP 将被设置成该段定义的字节数的总和；如果没有一个段的段组合属性设置成 stack，就必须用类似初始化 DS 和 ES 那样让 SS 指向你设置的堆栈段并让 SP 指向栈顶。

然后说说如何给 IP 赋初值。请注意程序举例中最后一条语句：

```
end   main
```

end 是一条伪指令，表示源程序到此为止，编译（汇编）程序不处理该语句之后的任何内容。另外，end 后面可以附带一个在代码段中已定义的标号，用以说明程序启动的偏移

量。如果源程序是一个独立程序或主模块，end 后面一定要带一个如上所述的标号，否则 end 之后就不能跟除注释之外的任何东西。所谓程序的运行，就是程序文件被操作系统引导到内存中，然后让指令指针 CS:IP 指向程序的起点，比如 main，具体地讲就是给 CS 和 IP 赋值；而 IP 所赋的值，就是由伪指令 end 指定的。

4）源程序基本框架

```
stack0  segment stack       ;堆栈段
        dw 40h dup(0)
stack0  ends

data    segment             ;数据段
        ;<变量定义>

data    ends

code    segment             ;代码段
        assume cs:code,ds:data,ss:stack0
main:   mov ax,data         ;ds指向数据段
        mov ds,ax

        ;<执行代码>


        mov ah,4ch          ;程序结束，退回DOS
        int 21h

code    ends
        end main
```

```
code    segment             ;代码段
        assume cs:code,ds:data,ss:stack0
myapp   proc far
main:   push ds             ;将指向int 20h的指针压栈
        xor ax,ax
        push ax

        mov ax,data         ;ds指向数据段
        mov ds,ax

        ;<执行代码>

        ret                 ;结束程序返回DOS
myapp   endp
code    ends
```

即把代码段做成一个被 DOS 调用的子程序的形式。

5. 编制汇编语言程序的完整过程

1）编程工具及经典过程

可使用任何纯文本编辑软件编辑汇编语言的源程序。我们接触较多的 Windows 平台纯文本的编辑软件有 UltraEdit、Edit Plus 等。

编译（汇编）软件，可使用微软的 MASM 或 Borland 公司的 TASM；连接软件可使用微软的 LINK.EXE 或 Borland 公司的 TLINK.EXE；至于调试软件，建议微软的 DEBUG 和 Borland 的 TD 都使用，各有千秋。下面以 Borland 公司的 TASM、TLINK、TD 及前述 Hello.asm 为例作扼要介绍程序编制的基本过程。

假定 TASM.EXE、TLINK.EXE、TD.EXE 及 Hello.asm 都在 D 盘名为“ASMTest”的文件夹中，则编译、连接及使用 TD 调试的命令行是这样的：

```
D:\ASMTest>TASM HELLO;

D:\ASMTest>TLINK HELLO;

D:\ASMTest>TD HELLO.EXE

D:\ASMTest>HELLO
```

如果在鼠标右键菜单上设置了“DOS 快速通道”功能，则通过操作 ASMTest 即可方便地进入 DOS 状态及编程文件夹：

2）用 UltraEdit 设置简易的汇编语言编程环境

点击 UltraEdit 主菜单的“高级/工具配置”，并在屏幕弹出的对话框中点“插入”：

以“连接”为例，设置如下：

（未完 看不懂）

3）DEBUG 常用命令简介

一．进入 DEBUG

在 DOS 提示符下键入 DEBUG，然后按 Enter 确认，屏幕显示：

二．和汇编学习有关的几个 DEBUG 命令

和汇编学习有关的几个 DEBUG 命令

命令字符	功　　能
?	显示 DEBUG 命令列表
q	退出 DEBUG
r	显示或设置寄存器值
d	显示默认或指定地址范围内存单元的值
e	从指定地址开始设置各字节单元的值
a	汇编命令
u	反汇编命令
g	运行指定地址开始的代码

DEBUG 只显示和接受 16 进制数

R 命令：

R 命令

从上图可见，没有参数的 r 命令将显示所有寄存器的内容；指定寄存器后 DEBUG 先显示指定寄存器的内容，然后用冒号“:”提示等待设置输入。上图记录了将寄存器 AX 设置成 1234H 的过程。

DEBUG 双字符状态表示的含义

标志位	含义	FLAG = 1		FLAG = 0	
OF	溢出(是/否)	OV	OVerflow	NV	Not oVerflow
DF	方向(减量/增量)	DN	DowN	UP	UP
IF	中断(允许/关闭)	EI	Enable Interrupt	DI	Disable Interrupt
SF	符号(负/正)	NG	NeGative	PL	PLus
ZF	零(是/否)	ZR	ZeRo	NZ	Not Zero
AF	辅助进位(是/否)	AC	Auxiliary Carry	NA	Not Auxiliary
PF	奇偶(是/否)	PE	Parity Even	PO	Parity Odd
CF	进位(是/否)	CY	CarrY	NC	Not Carry

习题：

2.用示意图说明下述为指令语句分配的字节数及各字节内容（假设数据段的段基址是 0ABCDH，且各变量依序从偏移地址 0 开始定义）。

```
VARA    EQU     10
VARB    DB      10
VARC    DW      1234H
VARD    DD      12345678H
VARE    DB      2 DUP(1,2),0
VARF    DW      VARB
VARG    DD      VARB
```

答：（注：VARA 为符号，因此未分配空间）

变量名称及对应的物理地址	内存单元中的内容（字节为单位）
VARB，0ABCDH	0AH
VARC，0ABCEH	34H
0ABCFH	12H
VARD，0ABD0H	78H
0ABD1H	56H
0ABD2H	34H
0ABD3H	12H

VARE，0ABD4H	01H
0ABD5H	02H
0ABD6H	01H
0ABD7H	02H
0ABD8H	00H
VARF，0ABD9H	00H （VARB 的偏移量）
0ABDAH	00H
VARG，0ABDBH	00H （VARB 的偏移量）
0ABDCH	00H
0ABDDH	0CDH （VARB 的段基值）
0ABDEH	0ABH

3. 说明下面代码中 N1、N2 和 N3 的异同：

```
N1      EQU     10
N2      DB      10
N3      DW      10
```

答：N1 为符号，引用 N1 等价于引用数值 10，汇编结束后，所有引用 N1 的位置都被替换为数值 10，汇编程序不会为 N1 分配空间；N2 为字节类型的变量，其长度为 1 个字节；N3 为字类型变量其长度为 2 个字节，N2 与 N3 的初值相同，引用 N2 与 N3 等价于引用这两个变量的段内偏移量。

① 定义一字符串变量且初值为“Hello,World!”。

```
STR1    DB    'Hello,World!'
```

DB 伪指令还支持用“字符串”表达式定义变量初值，如：

```
MSGHI       DB      'Hello,World!'
```

② 定义一字节型变量且初值为 0。

```
VAR1    DB    0
```

DW：字变量定义

③ 定义一未指定初值的 100 个单元的字型变量。

```
VAR1   DW  100  DUP(?)
```

④ 定义一值为 1000 的符号常量。

```
SYM1  EQU  1000
```

5. 对数据段定义的下述变量：

```
NUM1    DB      60
NUM2    EQU     50
NUM3    DB      20H
```

判断下述语句是否有错并指出是什么错误。

① MOV NUM2, AL

错误，NUM2 为符号常量，在指令中仅能作为源操作数，不能作为目的操作数，因为符号常量不会被分配空间，自然也就没有相应的地址。

② MOV AX, NUM3

错误，NUM3 为字节类型的变量，该指令试图将其内容传递到 16 位寄存器 AX 中，汇编程序无法明确应该传递到 AL 还是 AH，因此指令具有二义性，无法唯一解释。

③ MOV NUM3, NUM1

错误，8086/8088 指令系统的双操作数指令最多允许一个操作数使用存储器寻址方式。

④ MOV NUM1, NUM2

正确，这里 NUM2 是符号，在指令的源操作数处引用符号，对应其立即数字段。

⑤ MOV BX, AX+NUM2

错误，表达式中不能使用存储单元名称（包括寄存器、内存单元、端口），因为表达式是在汇编阶段完成运算的，而存储单元的内容必须要到程序执行阶段才能确定，因此包含存储单元的表达式无法在汇编阶段完成运算，自然也无法得到语法的允许。

⑥ MOV NUM1, AL and 0F0H

错误，原因同上题。

⑦ MOV AX, NUM3-NUM1

正确，在指令操作数位置引用两个变量之差，实质上是应用两个变量段内偏移量之差，即两个变量间的字节距离，而不是指两个变量内容之差。（注：在表达式中，两个变量间的运算仅允许减法，不允许加法，因为加法无意义）

6. 对于下面的数据定义,写出各条指令执行的结果:

```
FLDB DW 0A24FH
TABLE DB 32H,52,0C2H,213
TEA EQU WORD PTR TABLE
ARRAY DB 'ABCD'
COUNT EQU $-ARRAY
```

① MOV AX,FLDB

AX=0A24FH

② MOV BX,TEA

(BX)=0002H （假定 FLDB 的段内偏移量为 0）

Equ 是符号常量 TEA 表示偏移量
初始偏移量为 0000->下一个偏移量为 0002

③ MOV CH,TABLE+2

(CH)=0C2H

④ MOV DL,ARRAY

(DL)= ‘A’ （或 41H）

A 的 ascii 码为 65，16 进制为 41

⑤ MOV DH,COUNT

(DH)=04H

本题表达式中出现的$，是“编译（汇编）位置指针”，在编译（汇编）时将数据段的当前地址偏移量。

0002+2=0004

7. 假设数据段有下属变量定义：

```
INAME    DB  31 DUP(?)
ADDRESS DB  31 DUP(?)
CITY     DB  16 DUP(?)
ID       DB  1,2,1,6,8
```

① 用一条 MOV 语句将 INAME 的偏移量送入 SI。

```
MOV  SI, OFFSET INAME
```

② 用一条指令将 ID 的头两个字节的内容送入 BX。

```
MOV  BX, WORD PTR ID
```

③ 写出后三个字段的长度表达式。

CITY-ADDRESS

ID-CITY

$-ID

8. 在下述程序框架中分号“;”后面，描述同一行代码的功能。

```
stack0  segment stack 'stack'   ;
        dw 40h dup(0)           ;
stack0  ends

data    segment                 ;
        ;

data    ends

code    segment                 ;
        ;在此说明下一句
        assume  cs:code,ds:data,ss:stack0
main:   mov ax,data             ;
        mov ds,ax
        ;

        mov ah,4ch              ;
        int 21h

code    ends                    ;
        end main                ;
```

为突出重点，我们只要求程序框架堆栈段使用 STACK 组合类型，并是所以' STACK'类别。

STACK：组合类型 STACK 表示当前段是堆栈段，其组合情况与 PUBLIC 相同。

；定义一个段，名称为 stack0，segment 关键字后的 stack 关键字指明该段属性为堆栈段，运行该程序时，操作系统自动初始化 SS、SP，使用单引号括住的标识符为段的类别名称，运行程序时，操作系统会尽量将具有同一类别名的段分配在连续的内存空间内。

；分配 40H 个字的内存空间作为堆栈段空间

；定义数据段
；定义代码段
；使用 ASSUME 伪指令说明每个段寄存器默认指向的段（注：与段寄存器内容的初始化无关，DS、ES 初始化需使用指令完成；若指明堆栈段的堆栈属性，SS 的初始化由操作系统完成，若未指明堆栈属性，则也需要使用指令来完成；CS 的初始化由操作系统完成。

；此两条指令初始化 DS 段寄存器
；此两条指令使用系统调用结束程序运行，并将 CPU 控制权返还给操作系统。
；ENDS 结束代码段定义
；END 结束源程序，并给出程序中第一条指令所在的标号。

9. 编写一完整程序，该程序将数据段定义的字符串“Hello,World.”末尾的句号替换成感叹好“!”。

```
DATA    SEGMENT
VAR1 DB  ‘Hello, World.’
LEN     EQU $-VAR1
DATA ENDS

CODE SEGMENT
ASSUME DS:DATA
BEGIN:MOV  AX,DATA
        MOV DS,AX
        LEA SI,VAR1
        ADD SI,LEN-1
        MOV   BYTE   PTR[SI],
```

```
        '!'
        MOV AH,4CH
        INT 21H
CODE ENDS
END BEGIN
```

10. 编写一完整程序，分别在两个数据段 DSEG1 和 DSEG2 中各定义一个字型变量 X 和 Y 并计算两数差的绝对值。

```
YY1 SEGMENT
X   DW  4723H
Z   DW  ?
YY1 ENDS
YY2 SEGMENT
Y   DW  528AH
YY2 ENDS
CODE    SEGMENT
ASSUME  DS:YY1,ES:YY2
BEGIN:
    MOV AX,YY1
    MOV DS,AX
    MOV AX,YY2
    MOV ES,AX
    SUB AX,Y
    MOV Z,AX
    MOV AH,4CH
    INT 21H
CODE ENDS
END BEGIN
```

第七章 分支与循环设计

1. 无条件转移指令

无条件转移指令是指能够无条件改变程序执行流程，改变 CPU 顺序执行下一条指令的模式，而将程序流程转移到代码段指定位置的指令，该指定的转移位置称为目标地址。

功能是通过无条件修改 IP 寄存器的内容或修改 IP、CS 寄存器的内容来实现的。

可以分类为段内直接短转移、段内直接长转移、段内间接转移、段间直接转移、段间间接转移。

但在汇编指令系统中，它们使用相同的指令助记符“JMP”，即 jump 之意。

1) 段内直接转移

指令格式： JMP 标号

JMP NEAR PTR 标号

第一种格式中的标号与 JMP 指令位于同一代码段，并且定义的标号类型为 NEAR；第二种格式中的标号也与 JMP 指令位于同一代码段，但定义的标号类型允许为 FAR，PTR 运算符将标号类型在 JMP 指令中暂时的修改为 NEAR。

段内直接转移指令的机器指令由操作码字段、位移量字段（DISP 字段）构成

在汇编阶段（注意，指汇编程序将我们的源程序翻译为目标代码的阶段），汇编程序会生成机器指令中的位移量，生成方法为：DISP = 标号的段内偏移量 – JMP 指令后那条指令的段内偏移量 。

尽管这两个偏移量均是无符号数编码，由于标号的段内偏移量既可能大于 JMP 指令后那条指令的段内偏移量，也有可能小于它，因此所计算出的 DISP 可能为正数，也可能为负数，其编码一定是带符号数编码，即补码。

将 16 位的无符号偏移量看作 17 位的正数补码，最高位即第 16 位是不可见的，我们假定它总为 0，那么两个 17 位正数补码相减得到一个补码运算结果就不再是一件奇怪的事情了。

但是，与算术运算指令中的补码加、减运算类似，在计算 DISP 过程中也有可能出现补码运算溢出的情况，如果溢出发生，则汇编程序会在屏幕上报告一个语法错误，指出 JMP 指令的目标地址已超出其转移范围，程序员必须修改该错误后才能得到目标代码。

当所计算出的 DISP 位于-128~127 范围内时，它可用 8 位补码表示，此时汇编程序生成的机器指令为短转移指令，机器指令的操作码与位

移量字段总共占据 2 个字节。

而当所计算出的 DISP 超出-128~127 范围时，DISP 必须使用 16 位补码表示，此时汇编程序生成的机器指令为长转移指令，机器指令的操作码与位移量字段总共占据 3 个字节。段内直接转移最大范围为：向低地址端最大转移-32768 字节（约 32K），向高地址端最大转移 32767 字节（约 32K），若转移范围超出，则不能使用直接转移方式。

指令功能：(IP) + DISP ⇒ IP ，导致程序流程无条件地转移到目标地址，CPU 下一条即将执行的指令改变为目标地址处的指令。

指令功能中所述操作是在程序执行阶段，即执行 JMP 指令时完成的。指令功能中修改（IP）的操作是一个加法运算，与生成 DISP 的原理类似，在作加法时将（IP）理解为 17 位正数补码，并且由于汇编程序生成 DISP 时是保证了没有溢出发生的，因此 JMP 指令中使用加法来修改（IP）同样不会导致溢出产生，我们应将所得到的加法结果重新理解为 16 位的无符号偏移量。

2） 段内间接转移

指令格式： JMP 16 位通用寄存器名称

JMP 字类型的内存单元

指令功能：(16 位寄存器) ⇒ IP 或（字类型的内存单元）⇒ IP ，JMP 指令将 16 位通用寄存器或字类型内存单元中的数据解释为目标地址偏移量，并将其传送至 IP 寄存器保存，从而使程序流程转移至目标地址。段内间接转移中没有目标地址超出转移范围的情况发生。

例 7．1．1 无条件转移中的段内间接转移指令示例如下。

```
JMP   BX
```

该指令将（BX）解释为目标地址偏移量，并传送至 IP 保存。

```
JMP   [BX]0100H
```

该指令将逻辑地址为（DS）：（BX）+0100H 的字类型内存单元中的 16 位数据解释为目标地址偏移量，并传送至 IP 保存。

3） 段间直接转移

指令格式： JMP 标号

JMP FAR PTR 标号

第一种格式中的标号与 JMP 指令不在同一个代码段中，并且定义类型为 FAR；第二种 格式中的标号与 JMP 指令可能在不同代码段，也可能在同一代码段，但其定义类型可能为 NEAR，PTR 运算符将 JMP 指令中的标号类型暂时修改为 FAR。

指令功能： 标号的段内偏移量⇒ *IP*
标号所在段的段基值⇒ *CS*

段间直接转移没有任何转移范围限制，理论上可在 1M 字节内存空间中任意实施流程转移。

4） 段间间接转移

指令格式： JMP DWORD PTR 内存单元逻辑地址

指令功能：将双字内存单元中的低地址字解释为目标地址偏移量，传送至 IP 保存，将双字内存单元中的高地址字解释为目标地址段基值，传送至 CS 保存，从而使程序流程转移至目标地址。与段间直接转移类似，由于使用完整逻辑地址作为目标地址，转移范围在 1M 字节空间内不受限制。

例 7．1．2 段间间接转移指令示例如下。

```
.....
ADDR1   DD      L1
.....
JMP   DWORD PTR ADDR1
.....
```

2. 条件转移指令

在硬件上，多数条件转移指令所使用的判断条件都来自于状态标志位的当前状态，因此，我们可以称状态标志是实现分支、循环结构程序的硬件基础。

指令格式： JX NEAR 类型的段内标号

格式中 JX 表示抽象的助记符，可以使用任意具体的条件转移指令助记符来替换它。

1） 单标志位转移指令

单标志位转移指令是指判断条件基于单个标志位状态的条件转移指令。这种转移指令一般成对出现。

指令助记符	指令功能
JC	CF=1，转移至目标地址 CF=0，顺序执行
JNC	CF=1，顺序执行 CF=0，转移至目标地址
JO	OF=1，转移至目标地址 OF=0，顺序执行
JNO	OF=1，顺序执行 OF=0，转移至目标地址
JS	SF=1，转移至目标地址 SF=0，顺序执行
JNS	SF=1，顺序执行 SF=0，转移至目标地址

JZ / JE	ZF=1，转移至目标地址 ZF=0，顺序执行
JNZ / JNE	ZF=1，顺序执行 ZF=0，转移至目标地址
JP / JPE	PF=1，转移至目标地址 PF=0，顺序执行
JNP / JPO	PF=1，顺序执行 PF=0，转移至目标地址

2） JCXZ 指令

JCXZ 指令所判断的条件不是标志位的状态，而是 CX 寄存器中的数据，若 (CX)=0 则转移至目标地址，若 (CX) ≠ 0 则顺序执行。在 8086/8088 指令系统中，该指令是唯一不以标志位为判断条件的条件转移指令。

该指令的用途是配合循环控制指令形成计数循环结构，在进入循环体前用于判断计数器 CX 中的数据是否已经为零，如果已为零则不进入循环体，直接转移至循环出口。

3） 无符号数条件转移指令

无符号数条件转移指令的功能可以这样解释，将最近一条执行的 CMP（或 SUB）指令中两个操作数解释为无符号数编码，以 CMP 指令所影响的 CF、ZF 标志位来综合判断被减数与减数的大小关系，从而根据判断结果来决定是否实施流程转移。关于使用 CMP 指令判断无符号数大小关系的原理请参见关于第五章关于 CMP 指令的说明。读者应注意，**无符号数条件转移指令没有二意性，它固定地将 CMP 指令的操作数解释为无符号数**。因此，**在程序设计中，需要明确自己所使用的编码是无符号数之后，才能选择无符号数条件转移指令来实现分支或循环结构**。无符号数条件转移指令的助记符与功能解释如图 7.2.2 所示。图中的“A 数据”表示被减数，“B”数据表示减数。

指令助记符	指令功能
JA / JNBE	若 $A > B$，转移至目标地址 若 $A \le B$，顺序执行
JAE / JNB	若 $A \ge B$，转移至目标地址 若 $A < B$，顺序执行
JB / JNAE	若 $A < B$，转移至目标地址 若 $A \ge B$，顺序执行
JBE / JNA	若 $A \le B$，转移至目标地址 若 $A > B$，顺序执行

4） 带符号数条件转移指令

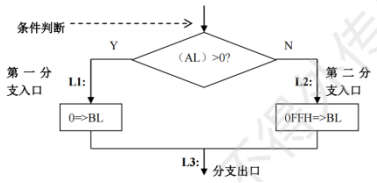
带符号数条件转移指令的功能可以这样解释，将最近一条执行的 CMP（或 SUB）指令中两个操作数解释为补码，以 CMP 指令所影响的 OF、SF、ZF 标志位来综合判断被减数与减数的大小关系，从而根据判断结果来决定是否实施流程转移。关于使用 CMP 指令判断补码大小关系的原理请参见关于第五章关于 CMP 指令的说明。读者应注意，**带符号数条件转移指令没有二意性，它固定地将 CMP 指令的操作数解释为补码**。因此，**在程序设计中，需要明确自己所使用的编码是补码之后，才能选择带符号数条件转移指令来实现分支或循环结构**。带符号数条件转移指令的助记符与功能解释如图 7.2.3 所示。图中的“A 数据”表示被减数，“B”数据表示减数。

指令助记符	指令功能
JG / JNLE	若 $A > B$，转移至目标地址 若 $A \le B$，顺序执行
JGE / JNL	若 $A \ge B$，转移至目标地址 若 $A < B$，顺序执行
JL / JNGE	若 $A < B$，转移至目标地址 若 $A \ge B$，顺序执行
JLE / JNG	若 $A \le B$，转移至目标地址 若 $A > B$，顺序执行

3. 分支程序设计

例 7．3．1 试设计一个程序片段，实现如下功能：将（AL）理解为补码，若（AL）>0，则将 BL 寄存器清 0；否则，将 BL 寄存器置为全 1。

流程图如下：

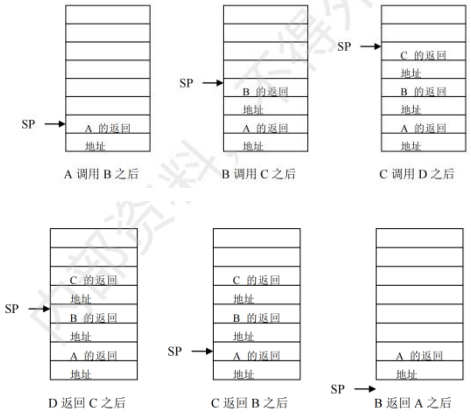


程序片段如下：

```
        CMP   AL，0          ；条件判断
        JG    L1
        JMP   L2
L1：    MOV   BL，0          ；第一分支入口
        JMP   L3             ；转移至分支结构出口
L2：    MOV   BL，0FFH       ；第二分支入口
L3：    ……                   ；分支结构出口
```

4.

例 8．1．2 假设主程序 A 在执行过程中调用了子程序 B，而子程序 B 在执行过程中调用了子程序 C，子程序 C 又调用了子程序 D，并且假设所有调用与返回都是段内调用与返回，那么，执行各子程序调用、返回指令时，堆栈中的变化如下所示（每个方格表示一个字节）。



乘法类型	隐含（目的）操作数	源操作数	乘积
字节乘 （Byte）	AL	8 位寄存器名 8 位数据地址	16 位数 存于 AX 中
字乘 （Word）	AX	16 位寄存器名 16 位数据地址	32 位数 存于 DX：AX 中
双字乘 （DWord）	EAX	32 位寄存器名 32 位数据地址	64 位数 存于 EDX：EAX 中

指令应用举例：

1．MUL　CL　；；无符号整数字节乘：（AL）x （CL），积存放在 AX 中。

2．IMU　BYTE PTR [BX] ；；有符号整数字节乘：（AL）x （[BX]），

；；积存放在 DX：AX 中。

3．MUL　DI　；；无符号整数字乘：（AX）x （DI），积存放在 DX：AX 中。

4．IMUL　EBX ；；有符号整数双字乘：（EAX）x （EBX），

；；积存放在 EDX：EAX 中。

5．MUL DWORD PTR[ECX] ；；无符号整数双字乘：（EAX）x （[ECX]），

；；积存放在 EDX：EAX 中

MUL　寄存器/内存操作数　；；　无符号整数乘法指令

IMUL　寄存器/ 内存操作数 ；；　有符号整数乘法指令

这两条指令具有以下特点：

1）双操作数指令 IMUL　Reg16，IMM

目的操作数 Reg16 是 16 位寄存器，源操作数为 16 位的立即数 IMM。16 位积存放在 Reg16 中。如果乘积超过 16 位，CF 和 OF 将置位。本条指令，既可用于有符号数乘，也可用于有符号数乘。比如指令　IMUL　DX，456。

2）　三操作数指令 IMUL　Reg16，Mem16，IMM

其中的第一个操作数必须是 16 位寄存器，第二个操作数必须是 16 位内存操作数 Mem16，第三个操作数是 16 位立即数 IMM。它可实现将第二个操作数和第三操作数相乘，乘积存放在第一操作数（16 位寄存器）中。

比如指令 IMUL BX，[SI]，35 则完成由 SI 指示内存单元字和立即数 35 相乘，结果存入 BX 中。

同样，当乘积大于 16 位整数范围时，CF 和 OF 将被置位。本指令也适用于无符号整数乘。

3）指令 IMUL　Reg，Reg/Mem

其中，第一个操作数是寄存器，它是目的操作数；第二个操作数是源操作数，它可以是寄存器操作数或内存操作数。目的操作数可以是 16 位或 32 位寄存器，源操作数必须与目的操作数的大小（Size）一致，即同为 16 位或 32 位操作数。乘积存放在目的操作数（第一操作数寄存器）中。比如：

IMUL　DX，25 ；16 位数相乘，（DX）x　25　→ DX

IMUL　ECX，MULD，25；32 位数相乘，（MULD）x　25　→ ECX

IMUL　BX，CX；　16 位数相乘，（BX）x （CX）　→ BX

同乘法指令相类似，80x86 汇编同样有无符号整数除指令（DIV）和有符号整数除指令（IDIV）。由于是整数除法，除法结果包括了商和余数。

1．指令格式

DIV　Regs/Mem

IDIV　Regs/Mem

除数是指令中的操作数，可以是寄存器操作数（Regs）或内存操作数（Mem）。它的大小（Size），表明了本指令是进行“字节除法”、“字除法”或“双字除法”。

2．被除数、除数和商

除法中的被除数为隐含操作数，它与除数、商和余数的关系，见表 9.2。

除法类型	被除数	除数	商和余数
字节除法	AX	8 位寄存器或内存操作数	商：AL，　余数：AH
字除法	DX：AX	16 位寄存器或内存操作数	商：AX，　余数：DX
双字除法	EDX：EAX	32 位寄存器或内存操作数	商：EAX，　余数：EDX

从表中可见，在字节除法中，AH 最好不要作为除数；在字除法中，DX 和 AX 最好不要作为除数；在双字除法中，EDX 和 EAX 最好不要作为除数，因为在除法操作结束后，它们会被占用。

3．　对于无符号整数除法来说，在除法操作之前，首先应按表 9.2 的要求，准备好被除数。对于除数是 8 位（字节除）、16 位（字除）或 32 位（双字除）整数时，相应的被除数必需是 16 位、32 位或 64 位。若不符合这个要求，可采将被除数高位字节、字或双字填零的办法，生成 16 位、32 位或 64 位的被除数。比如无符号整数除：（AL）＝124 除以 23 可用以下指令完成：

3．　对于无符号整数除法来说，在除法操作之前，首先应按表 9.2 的要求，准备好被除数。对于除数是 8 位（字节除）、16 位（字除）或 32 位（双字除）整数时，相应的被除数必需是 16 位、32 位或 64 位。若不符合这个要求，可采将被除数高位字节、字或双字填零的办法，生成 16 位、32 位或 64 位的被除数。比如无符号整数除：（AL）＝124 除以 23 可用以下指令完成：

XOR　AH，AH　　；；　0 → AH，使被除数为 AX

MOV　BL，23

DIV　BL

执行结果：（AL）＝　5　（商），（AH）＝　9（余数）

指令	功能说明
CBW	将 AL 中符号位扩展到 AH，　AX 成为有符号的字
CWD	将 AX 中符号位扩展到 DX，　DX：AX 成为有符号的双字
CWDE	扩展 AX 中符号位，　EAX 变成有符号的双字
CDQ	将 EAX 中符号位扩展到 EDX，EDX：EAX 成为有符号 64 位数

比如，有符号整数除：（AX）＝－8124 除以 23 可用以下指令完成：

CDW　　；扩展有符号数符号位到 DX，形成有符号双字 DX：AX

MOV　CX，23

IDIV　CX

执行结果：（AX）＝－353　（商），（DX）＝－5（余数）

5．需要特别说明的是，无论在执行 DIV 或 IDIV 指令中，是除数等于 0，或是商太大或太小，超过存放商的单元所能表示范围，都会产生除法溢出中断。比如，（AX）＝1321 除以 5，商为 264，显然超过 AL 所能存放（表示）的最大整数 255，故将发生除法溢出中断。

又如，被除数 6553 除以－5，若使用字节除，商为－1310，余 3。显然－1310 超过了一个字节所能表示的最小的负数（－128），故也会发生除法溢出中断。

1．非组合型（Unpacked）BCD 码

一位非组合型 BCD 码占用一个字节的低 4 位（D3 D2 D1 D0），高 4 位用 0 填充。也就是说，一位 BCD 码占用一个字节。如要存放一个两位十进制数，需占用 2 个字节。通常，高地址字节存放十位数，低地址字节存放个位数。

2．组合型（Packed）BCD 码

一个组合型 BCD 码由两位 BCD 码组成，占用一个字节。其中，一个 BCD 码位占用字节的低 4 位（D3 D2 D1 D0），另一个 BCD 码位则占用字节的高 4 位（D7 D6 D5 D4）。如果我们将一个组合型 BCD 码看成是一个十进制数，那么高 4 位可以认为是十位数，而低 4 位则可以认为是个位数。这也就是说，一个字节可存放 0 － 99 间的 BCD 码值，或十进制数 0 －99。显然，若要表示 4 位十进制数，就要占用 2 个字节，低地址字节则存放十位和个位，相邻的高地址字节存放千位和百位。

我们如果要在数据段中定义非组合型 BCD 和组合型 BCD 格式的十进制数 89，67，23，9 时，分别使用以下语句：

```
UnBCD   DB   09，08，07，06，03，02，09，00；定义非组合型 BCD
BCDA    DB   89H，67H，23H，09H；定义组合型 BCD
```

又如，将 65 以组合型 BCD 码形式存入 CL，以非组合型 BCD 码形式存入 DX，使用指令：

```
MOV   CL，65H
MOV   DX，0605H
```

BCD 码的加减法指令有：

非组合型 BCD 码加法调整指令 AAA（ASCII Adjust after Adition）

非组合型 BCD 码减法调整指令 AAS（ASCII Adjust after Subtraction ）

组合型 BCD 码加法调整指令 DAA（Decimal Adjust after Adition）

组合型 BCD 码减法调整指令 DAS（Decimal Adjust after Subtraction）

这些指令都有一个共同的特点：它们都是用在加指令（ADD、ADC）和减指令（SUB、

比如： 计算 BCD 码 4＋9：

```
MOV   AX，9
ADD   AL，4        ；；  完成加后，（AL）＝ 0DH
AAA                ；；调整 AL 的值：由于（AL）>9，所以(AL)+6→AL，
                   ；；（AH）+1→AH，AL 中高 4 位清 0，CF 置 1。
```

最后（AH）＝ 1，（AL）＝ 3，CF＝1，即存放在 AX 中的非组合型 BCD 码 0103H，应看作是十进制数 13。

由此可见，两个 1 位 BCD 码（或数字的 ASCII）在完成相加（比如 ADD、ADC 指令）操作后，再用 AAA 指令对 AL 中的值进行调整，便可将结果调整为 BCD 码（0－9）。调整的方法是：若（AL）> 9 或 AF 置 1，则 AH 加 1，CF 和 AF 置 1，AL 高 4 位清 0；否则，CF 和 AF 将清 0。相加和以非组合型 BCD 格式存放在 AX 中。

又例如：

```
MOV   AX，0209H    ；；注意（AH）＝2
```



```
ADD   AL，5
AAA                ；；调整后，（AX）＝0304H，可以认为是计算 29＋5 的和。
```

2. 组合型 BCD 码加法调整指令 DAA

指令格式： DAA

DAA 用于对两个组合型 BCD 码进行相加的加法指令之后，对存放在 AL 中的和进行调整。调整后的 BCD 码仍保存在 AL 中。注意，加法指令中的目的操作数必须是寄存器 AL。

调整准则：

1） 若 AL 的低 4 位>9 或向高 4 位有进位（AF＝1），则（AL）＋6→AL，且 1→AF；

2） 若 AL 的高 4 位>9 或向更高位（比如百位）有进位（CF＝1），则（AL）＋60H →AL，且 1→AF（高 4 位>9 时），1→CF；

3） 若不属于 1）、2）情况，则 0→CF ，0→AF。

比如实现十进制数 4534＋287 的指令段如下，和存放在 DX 中，程序未考虑相加可能向更高位进位的情况。

```
MOV   AL，34H     ；；计算低字节和（十位和个位）
ADD   AL，87H     ；；  （AL）＝BBH
DAA               ；；  调整：（AL）＝21H，（CF）＝1，（AF）＝1
MOV   DL，AL      ；；   保存低位 BCD 码 → DL
MOV   AL，45H     ；；  计算高字节和（千位和百位）
ADC    AL，02H    ；；  低 2 位向高位有进位，（AL）＝48H
DAA               ；；调整，（CF）＝0，（AF）＝0
MOV   DH，AL      ；；保存高位 BCD 码 → DH
；；  （DX）＝4821H
```

3. 非组合型 BCD 码减法调整指令 AAS

非组合型 BCD 码减法调整指令 AAS，同样又称为 ASCII 减法调整指令。它不但可以用于对 1 位 BCD 码相减后的结果进行调整，也可用于两个数字的 ASCII 相减后的结果进行调整。

指令格式： AAS

AAS 用在对两位非组合型 BCD 码相减的减法指令后，通过对存放在 AL 中的差的调整，使 AL 中的值为非组合型 BCD（0－9）。调整准则是：

1） 若（AL）＞9 或 AF＝1（向高位有借位），则（AL）－6→AL，（AH）－1→AH，1→CF 和 1→AF，AL 高 4 位清 0；

2） 若不符合 1）的条件，则 CF 和 AF 清 0。

```
MOV   AX，0103H
SUB   AL，4        ；；（AL）－4＝－1（即 FFH）
AAS                ；；调整 AL：（AL）－6→AL，（AH）－1→AH，AL 高 4
                   ；；位清 0，1→CF，1→AF，差（AX）=0009
```

本例类似于 13－4＝9。

```
MOV   AX，04
SUB   AL，08
AAS
```

（AX）＝FF06H，AF＝1，CF＝1 。由于 CF＝1，表示是向高位有借位，或结果为负。

比如计算 83－38

```
MOV   AX，3883H
SUB   AL，AH    ；；83H－38H＝4BH ，即（AL）＝4BH
DAS             ；；对 AL 进行调整，形成组合型 BCD，（AL）＝45H，
                ；；CF ＝1，AF＝1 。表示差为 45。
```

2．在数据段或寄存器中，定义组合型 BCD 码时，字节的低 4 位存放低位 BCD 码；字节的高 4 位存放高位 BCD 码；每一个组合型 BCD 码都必须定义成 Hex。

例如，定义组合型 BCD 码数据 35，78，97，45 等时，用以下伪指令：

```
DB    35H，78H，97H，45H
MOV   AL，97H
```

若用指令 DB 35，78，97，45 和 MOV AL，45，在汇编后，存放在内存中的数据实际上是 23H，4EH，61H，2DH，它们显然不是我们所需要的组合型 BCD 值。

3．对于非组合型 BCD 码加减操作，一次只能完成一位数加减；对于组合型 BCD 码加减操作，一次只能完成两位数加减。也就是说，它们每次进行只能字节加减运算。

4．无论是非组合型还是组合型 BCD 码，加减运算后，被调整的对象总是 AL 中的内容。

对于非组合型 BCD 码的调整中，进位或借位（CF＝1），都会使（AH）受影响，即（AH）±1→AH。

对于组合型 BCD 码的调整中，高位 BCD 码的进位或借位，只会使 1→CF，不会影响 AH 中的值。

和汇编学习有关的几个 DEBUG 命令

命令字符	功　　能
?	显示 DEBUG 命令列表
q	退出 DEBUG
r	显示或设置寄存器值
d	显示默认或指定地址范围内存单元的值
e	从指定地址开始设置各字节单元的值
a	汇编命令
u	反汇编命令
g	运行指定地址开始的代码