单片机第八课（寻址方式与指令系统）

通过前面的学习，我们已经了解了单片机内部的结构，并且也已经知道，要控制单片机，让它为我们干学，要用指令，我们已学了几条指令，但很零散，从现在开始系统地学习 8051 的指令部份。

一、概述

1、指令的格式

我们已知，要让计算机做事，就得给计算机以指令，并且我们已知，计算机很“笨”，只能懂得数字，如前面我们写进机器的 75H，90H，00H 等等，所以指令的第一种格式就是机器码格式，也说是数字的形式。但这种形式实在是为难我们人了，太难记了，于是有另一种格式，助记符格式，如 MOV P1，#0FFH，这样就好记了。 这两种格式之间的关系呢，我们不难理解，本质上它们完全等价，只是形式不一样而已。

2、汇编

我们写指令使用汇编格式，而计算机只懂机器码格式，所以要将我们写的汇编格式的指令转换为机器码格式，这种转换有两种方法：手工汇编和机器汇编。手工汇编实际上就是查表，因为这两种格式纯粹是格式不同，所以是一一对应的，查一张表格就行了。不过手工查表总是嫌麻烦，所以就有了计算机软件，用计算机软件来替代手工查表，这就是机器汇编。

二、寻址

让我们先来复习一下我们学过的一些指令：MOV P1，#0FFH，MOV R7，#0FFH 这些指令都是将一些数据送到相应的位置中去，为什么要送数据呢？第一个因为送入的数可以让灯全灭掉，第二个是为了要实现延时，从这里我们可以看出来，在用单片机的编程语言编程时，经常要用到数据的传递，事实上数据传递是单片机编程时的一项重要工作，一共有 28 条指令（单片机共 111 条指令）。下面我们就从数据传递类指令开始吧。

分析一下 MOV P1，#0FFH 这条指令，我们不难得出结论，第一个词 MOV 是命令动词，也就是决定做什么事情的，MOV 是 MOVE 少写了一个 E，所以就是“传递”，这就是指令，规定做什么事情，后面还有一些参数，分析一下，数据传递必须要有一个“源”也就是你要送什么数，必须要有一个“目的”，也就是你这个数要送到什么地方去，显然在上面那条指令中，要送的数（源）就是 0FFH，而要送达的地方（目的地）就是 P1 这个寄存器。在数据传递类指令中，均将目的地写在指令的后面，而将源写在最后。

这条指令中，送给 P1 是这个数本身，换言之，做完这条指令后，我们可以明确地知道，P1 中的值是 0FFH，但是并不是任何时候都可以直接给出数本身的。例如，在我们前面给出的延时程序例是这样写的：

```
MAIN:  SETB P1.0        ;（1）
       LCALL DELAY ;（2）
         CLR P1.0         ;（3）
```

```
      LCALL DELAY     ；（4）
        AJMP MAIN       ；（5）
；以下子程序
DELAY：  MOV R7，#250     ；（6）
D1：  MOV R6，#250     ；（7）
D2：  DJNZ R6，D2       ；（8）
     DJNZ R7，D1      ；（9）
     RET              ；（10）
     END              ；（11）
```

表 1

```
 MAIN：  SETB P1.0         ；（1）
      MOV 30H，#255
       LCALL DELAY ；
       CLR P1.0          ；（3）
       MOV 30H,#200
       LCALL DELAY     ；（4）
       AJMP MAIN       ；（5）
；以下子程序
DELAY：  MOV R7，30H     ；（6）
D1：  MOV R6，#250     ；（7）
D2：  DJNZ R6，D2       ；（8）
     DJNZ R7，D1      ；（9）
     RET              ；（10）
     END              ；（11）
```

这样一来，我每次调用延时程序延时的时间都是相同的（大致都是 0.13S)，如果我提出这样的要求：灯亮后延时时间为 0.13S 灯灭，灯灭后延时 0.1 秒灯亮，如此循环，这样的程序还能满足要求吗？不能，怎么办？我们可以把延时程序改成这样(见表 2)：调用则见表 2 中的主程，也就是先把一个数送入 30H，在子程序中 R7 中的值并不固定，而是根据 30H 单元中传过来的数确定。这样就可以满足要求。

从这里我们可以得出结论，在数据传递中要找到被传递的数，很多时候，这个数并不能直接给出，需要变化，这就引出了一个概念：如何寻找操作数，我们把寻找操作数所在单元的地址称之为寻址。在这里我们直接使用数所在单元的地址找到了操作数，所以称这种方法为直接寻址。除了这种方法之外，还有一种，如果我们把数放在工作寄存器中，从工作寄存器中寻找数据，则称之为寄存器寻址。例：MOV A，R0 就是将 R0 工作寄存器中的数据送到累加器 A 中去。提一个问题：我们知道，工作寄存器就是内存单元的一部份，如果我们选择工作寄存器组 0，则 R0 就是 RAM 的 00H 单元，那么这样一来，MOV A，00H，和 MOV A，R0 不就没什么区别了吗？为什么要加以区分呢？的确，这两条指令执行的结果是完全相同的，都是将 00H 单元中的内容送到 A 中去，但是执行的过程不同，执行第一条指令需要 2 个周期，而第二条则只需要 1 个周期，第一条指令变成最终的目标码要两个字节（E5H 00H），而第二条则只要一个字节（E8h）就可以了。

这么斤斤计较！不就差了一个周期吗，如果是 12M 的晶振的话，也就 1 个微秒时间了，一个字节又能有多少？

不对，如果这条指令只执行一次，也许无所谓，但一条指令如果执行上 1000 次，就是 1 毫秒，如果要执行 1000000 万次，就是 1S 的误差，这就很可观了，单片机做的是实时控制的

事，所以必须如此“斤斤计较”。字节数同样如此。

再来提一个问题，现在我们已知，寻找操作数可以通过直接给的方式（立即寻址）和直接给出数所在单元地址的方式（直接寻址），这就够了吗？

看这个问题，要求从 30H 单元开始，取 20 个数，分别送入 A 累加器。

就我们目前掌握的办法而言，要从 30H 单元取数，就用 MOV A，30H，那么下一个数呢？是 31H 单元的，怎么取呢？还是只能用 MOV A，31H，那么 20 个数，不是得 20 条指令才能写完吗？这里只有 20 个数，如果要送 200 个或 2000 个数，那岂不要写上 200 条或 2000 条命令?这未免太笨了吧。为什么会出现这样的状况？是因为我们只会把地址写在指令中，所以就没办法了，如果我们不是把地址直接写在指令中，而是把地址放在另外一个寄存器单元中，根据这个寄存器单元中的数值决定该到哪个单元中取数据，比如，当前这个寄存器中的值是 30H，那么就到 30H 单元中去取，如果是 31H 就到 31H 单元中去取，就可以解决这个问题了。怎么个解决法呢？既然是看的寄存器中的值，那么我们就可以通过一定的方法让这里面的值发生变化，比如取完一个数后，将这个寄存器单元中的值加 1，还是执行同一条指令，可是取数的对象却不一样了，不是吗。通过例子来说明吧。

```
MOV R7，#20
     MOV R0，#30H
LOOP：MOV A，@R0
     INC R0
     DJNZ R7,LOOP
```

这个例子中大部份指令我们是能看懂的，第一句，是将立即数 20 送到 R7 中，执行完后 R7 中的值应当是 20。第二句是将立即数 30H 送入 R0 工作寄存器中，所以执行完后，R0 单元中的值是 30H，第三句，这是看一下 R0 单元中是什么值，把这个值作为地址，取这个地址单元的内容送入 A 中，此时，执行这条指令的结果就相当于 MOV A，30H。第四句，没学过，就是把 R0 中的值加 1，因此执行完后，R0 中的值就是 31H，第五句，学过，将 R7 中的值减 1，看是否等于 0，不等于 0，则转到标号 LOOP 处继续执行，因此，执行完这句后，将转去执行 MOV A，@R0 这句话，此时相当于执行了 MOV A，31H（因为此时的 R0 中的值已是 31H 了），如此，直到 R7 中的值逐次相减等于 0，也就是循环 20 次为止，就实现了我们的要求：从 30H 单元开始将 20 个数据送入 A 中。

这也是一种寻找数据的方法，由于数据是间接地被找到的，所以就称之为间址寻址。注意，在间址寻址中，只能用 R0 或 R1 存放等寻找的数据。

单片机教程第九课：数据传递指令

数据传递类指令
1） 以累加器为目的操作数的指令
MOV A，Rn
MOV A，direct
MOV A，@Ri
MOV A，#data
第一条指令中，Rn 代表的是 R0-R7。第二条指令中，direct 就是指的直接地址，而第三条指令中，就是我们刚才讲过的。第四条指令是将立即数 data 送到 A 中。
下面我们通过一些例子加以说明：
MOV A，R1 ；将工作寄存器 R1 中的值送入 A，R1 中的值保持不变。
MOV A,30H ；将内存 30H 单元中的值送入 A，30H 单元中的值保持不变。
MOV A,@R1 ；先看 R1 中是什么值，把这个值作为地址，并将这个地址单元中的值送入 A 中。如执行命令前 R1 中的值为 20H，则是将 20H 单元中的值送入 A 中。
MOV A,#34H ；将立即数 34H 送入 A 中，执行完本条指令后，A 中的值是 34H。
2）以寄存器 Rn 为目的操作的指令
MOV Rn,A
MOV Rn,direct
MOV Rn,#data
这组指令功能是把源地址单元中的内容送入工作寄存器，源操作数不变。
单片机指令（二）
数据传递类指令
（3）以直接地址为目的操作数的指令 MOV direct,A 例： MOV 20H,AMOV direct,Rn MOV 20H,R1
MOV direct1,direct2 MOV 20H,30H
MOV direct,@Ri MOV 20H,@R1
MOV direct,#data MOV 20H,#34H
（4）以间接地址为目的操作数的指令 MOV @Ri,A 例：MOV @R0,AMOV @Ri,direct MOV @R1,20H
MOV @Ri,#data MOV @R0,#34H
（5）十六位数的传递指令 MOV DPTR，#data168051 是一种 8 位机，这是唯一的一条 16 位立即数传递指令，其功能是将一个 16 位的立即数送入 DPTR 中去。其中高 8 位送入 DPH，低 8 位送入 DPL。例：MOV DPTR，#1234H，则执行完了之后 DPH 中的值为 12H，DPL 中的值为 34H。反之，如果我们分别向 DPH，DPL 送数，则结果也一样。如有下面两条指令：MOV DPH，#35H，MOV DPL，#12H。则就相当于执行了 MOV DPTR，#3512H。综合练习：
给出每条指令执行后的结果
MOV 23H,#30HMOV 12H,#34H
MOV R0,#23H
MOV R7,#22H
MOV R1,12H

MOV A,@R0
MOV 34H,@R1
(23h)=30h
(12h)=34h
(R0)=23H
(R7)=22H
(R1)=12H
(A)=30H
(34H)=34H
MOV 45H,34H
MOV DPTR,#6712H
MOV 12H,DPH
MOV R0,DPL
MOV A,@R0
(45H)=34H
(DPTR)=6712H
(12H)=67H
(R0)=12H
(A)=67H

说明：用括号括起来代表内容，如（23H）则代表内部 RAM23H 单元中的值，（A）则代表累加器 A 单元中的值。

上机练习：

进入 DOS 状态，进入 WAVE 所在的目录，例 D:\WAVE

键入 MCS51，出现如下画面

图 1

按 File->Open，出现对话框后，在 Name 处输入一个文件名（见图 2），如果是下面列表中已存在的，则打开这个文件，如果不存在这个文件，则新建一个文件（见图 3）

图 2

在空白处将上面的程序输入。见图 4。用 ALT+A 汇编通过。用 F8 即可单步执行，在执行过程中注意观察屏幕左边的工作寄存器及 A 累加器中的值的变化。

图 4

内存中值的变化在此是看不到的，可以用如下方法观察（看图 5）：将鼠标移到 DATA，双击，则光标进入此行，此时可以键盘上的上下光标键上下翻动来观察内存值的变化。本行的最前面 DATA 后面的数据代表的是“一段”的开始地址，如现在为 20H，再看屏幕的最上方，数字从 0 到 F，显示两者相加就等于真正的地址值，如现在图上所示的内存 20H、21H、22H、23H 中的值分别是 FBH 、0EH、E8H、30H。

图 56、当运行完程序后，即进入它的反汇编区，不是我们想要的东西。为了再从头开始，可以用 CTRL+F2 功能键复位 PC 值。注意此时不会看到原来的窗口，为看到原来的窗口，请用 ALT+4 或 ALT+5 等来切换。当然以上操作也可以菜单进行。CTRL+F2 是程序复位，用 RUN 菜单。窗口用 WINDOWS 菜单。

此次大家就用用熟这个软件吧，说实话，我并不很喜欢它，操作起来不方便，但给我的机器只能上这个，没办法，下次再给网友单独介绍一个好一点的吧。

本页图片较多，如果大家无法忍受它的等待，请下载

单片机教程第十课数据传递类指令指令

累加器 A **与片外** RAM **之间的数据传递类指令**

MOVX A,@Ri

MOVX @Ri,A

MOVX A,@DPTR

MOVX @DPTR,A

说明：

1）**在** 51 **中，与外部存储器** RAM **打交道的只可以是** A **累加器。所有需要送入外部** RAM **的数据必需要通过** A **送去，而所有要读入的外部** RAM **中的数据也必需通过** A **读入。在此我们可以看出内外部** RAM **的区别了，内部** RAM **间可以直接进行数据的传递，而外部则不行，比如，要将外部** RAM **中某一单元（设为** 0100H **单元的数据）送入另一个单元（设为** 0200H **单元），也必须先将** 0100H **单元中的内容读入** A，**然后再送到** 0200H **单元中去。**

要读或写外部的 RAM，**当然也必须要知道** RAM **的地址，在后两条指令中，地址是被直接放在** DPTR **中的。而前两条指令，由于** Ri（**即** R0 **或** R1）**只是一个** 8 **位的寄存器，所以只提供低** 8 **位地址。因为有时扩展的外部** RAM **的数量比较少，少于或等于** 256 **个，就只需要提供** 8 **位地址就够了。**

使用时应当首先将要读或写的地址送入 DPTR **或** Ri **中，然后再用读写命令。**

例：将外部 RAM **中** 100H **单元中的内容送入外部** RAM **中** 200H **单元中。**

MOV DPTR，#0100H

MOVX A，@DPTR

MOV DPTR,#0200H

MOVX @DPTR,A

程序存储器向累加器 A **传送指令**

MOVC A，@A+DPTR　　**本指令是将** ***ROM*** **中的数送入** A **中。本指令也被称为查表指令，常用此指令来查一个已做好在** ***ROM*** **中的表格**　　**说明：**

此条指令引出一个新的寻址方法：变址寻址。本指令是要在 ROM **的一个地址单元中找出数据，显然必须知道这个单元的地址，这个单元的地址是这样确定的：在执行本指令立脚点** DPTR **中有一个数，** A **中有一个数，执行指令时，将** A **和** DPTR **中的数加起为，就成为要查找的单元的地址。**

查找到的结果被放在 A **中，因此，本条指令执行前后，** A **中的值不一定相同。**

例：有一个数在 R0 **中，要求用查表的方法确定它的平方值（此数的取值范围是** 0-5）

MOV DPTR，#TABLE

MOV A，R0

MOVC A，@A+DPTR

TABLE: DB 0,1,4,9,16,25

设 R0 **中的值为** 2，**送入** A **中，而** DPTR **中的值则为** TABLE，**则最终确定的** ROM **单元的地址就是** TABLE+2，**也就是到这个单元中去取数，取到的是** 4，**显然它正是** 2 **的平方。其它数据也可以类推。**

标号的真实含义：从这个地方也可以看到另一个问题，我们使用了标号来替代具体的单元地址。事实上，标号的真实含义就是地址数值。在这里它代表了，0，1，4，9，16，25 **这几个数据在** ROM **中存放的起点位置。而在以前我们学过的如** LCALL DELAY **指令中，**

DELAY 则代表了以 DELAY 为标号的那段程序在 ROM 中存放的起始地址。事实上，CPU 正是通过这个地址才找到这段程序的。

可以通过以下的例子再来看一看标号的含义：

```
MOV DPTR，#100H
MOV A，R0
MOVC A，@A+DPTR
ORG 0100H.
DB 0,1,4,9,16,25
```

如果 R0 中的值为 2，则最终地址为 100H+2 为 102H，到 102H 单元中找到的是 4。这个可以看懂了吧？

那为什么不这样写程序，要用标号呢？不是增加疑惑吗？

如果这样写程序的话，在写程序时，我们就必须确定这张表格在 ROM 中的具体的位置，如果写完程序后，又想在这段程序前插入一段程序，那么这张表格的位置就又要变了，要改 ORG 100H 这句话了，我们是经常需要修改程序的，那多麻烦，所以就用标号来替代，只要一编译程序，位置就自动发生变化，我们把这个麻烦事交给计算机（指 PC 机）去做了。

堆栈操作

```
PUSH direct
POP direct
```

第一条指令称之为推入，就是将 direct 中的内容送入堆栈中，第二条指令称之为弹出，就是将堆栈中的内容送回到 direct 中。推入指令的执行过程是，首先将 SP 中的值加 1，然后把 SP 中的值当作地址，将 direct 中的值送进以 SP 中的值为地址的 RAM 单元中。例：

```
MOV SP，#5FH
MOV A，#100
MOV B，#20
PUSH ACC
PUSH B
```

则执行第一条 PUSH ACC 指令是这样的：将 SP 中的值加 1，即变为 60H，然后将 A 中的值送到 60H 单元中，因此执行完本条指令后， 内存 60H 单元的值就是 100，同样，执行 PUSH B 时，是将 SP+1，即变为 61H，然后将 B 中的值送入到 61H 单元中，即执行完本条指令后，61H 单元中的值变为 20。

POP 指令的执行是这样的，首先将 SP 中的值作为地址，并将此地址中的数送到 POP 指令后面的那个 direct 中，然后 SP 减 1。

接上例：

```
POP B
POP ACC
```

则执行过程是：将 SP 中的值（现在是 61H）作为地址，取 61H 单元中的数值（现在是 20），送到 B 中，所以执行完本条指令后 B 中的值是 20，然后将 SP 减 1，因此本条指令执行完后，SP 的值变为 60H，然后执行 POP ACC，将 SP 中的值（60H）作为地址，从该地址中取数（现在是 100），并送到 ACC 中，所以执行完本条指令后，ACC 中的值是 100。

这有什么意义呢？ACC 中的值本来就是 100，B 中的值本来就是 20，是的，在本例中，的确没有意义，但在实际工作中，则在 PUSH B 后往往要执行其他指令，而且这些指令会把 A 中的值，B 中的值改掉，所以在程序的结束，如果我们要把 A 和 B 中的值恢复原值，那么这些指令就有意义了。

还有一个问题，如果我不用堆栈，比如说在 PUSH ACC 指令处用 MOV 60H，A，在 PUSH

B 处用指令 MOV 61H，B，然后用 MOV A，60H，MOV B，61H 来替代两条 POP 指令，不是也一样吗？是的，从结果上看是一样的，但是从过程看是不一样的，PUSH 和 POP 指令都是单字节，单周期指令，而 MOV 指令则是双字节，双周期指令。更何况，堆栈的作用不止于此，所以一般的计算机上都设有堆栈，而我们在编写子程序，需要保存数据时，通常也不采用后面的方法，而是用堆栈的方法来实现。

例：写出以下程序的运行结果

```
MOV 30H，#12
MOV 31H，#23
PUSH 30H
PUSH 31H
POP 30H
POP 31H
```

结果是 30H 中的值变为 23，而 31H 中的值则变为 12。也就两者进行了数据交换。从这个例子可以看出：使用堆栈时，入栈的书写顺序和出栈的书写顺序必须相反，才能保证数据被送回原位，否则就要出错了。

作业：在 MCS51 下执行上面的例程，注意观察内存窗口和堆栈的变化。

单片机教程第十一课：算术运算类指令

不带进位位的加法指令

ADD A,#DATA ;**例**：ADD A，#10H

ADD A,direct ;**例**：ADD A，10H

ADD A,Rn ;**例**：ADD A，R7

ADD A,@Ri ;**例**：ADD A，@R0

用途：**将** A **中的值与其后面的值相加**，**最终结果否是回到** A **中**。

例：MOV A，#30H

ADD A，#10H

则执行完本条指令后，A **中的值为** 40H。

下面的题目自行练习

MOV 34H，#10H

MOV R0，#13H

MOV A，34H

ADD A，R0

MOV R1，#34H

ADD A，@R1

带进位位的加法指令

ADDC A，Rn

ADDC A,direct

ADDC A,@Ri

ADDC A,#data

用途：**将** A **中的值和其后面的值相加**，**并且加上进位位** C **中的值**。

说明：**由于** 51 **单片机是一种** 8 **位机**，**所以只能做** 8 **位的数学运算**，**但** 8 **位运算的范围只有** 0-255，**这在实际工作中是不够的**，**因此就要进行扩展**，**一般是将** 2 **个** 8 **位的数学运算合起来**，**成为一个** 16 **位的运算**，**这样**，**可以表达的数的范围就可以达到** 0-65535。**如何合并呢**？**其实很简单**，**让我们看一个** 10 **进制数的例子**：

66+78。

这两个数相加，**我们根本不在意这的过程**，**但事实上我们是这样做的**：**先做** 6+8（**低位**），**然后再做** 6+7，**这是高位**。**做了两次加法**，**只是我们做的时候并没有刻意分成两次加法来做罢了**，**或者说我们并没有意识到我们做了两次加法**。**之所以要分成两次来做**，**是因为这两个数超过了一位数所能表达的范置**（0-9）。

在做低位时产生了进位，**我们做的时候是在适当的位置点一下**，**然后在做高位加法是将这一点加进去**。**那么计算机中做** 16 **位加法时同样如此**，**先做低** 8 **位的**，**如果两数相加产生了进位**，**也要“点一下”做个标记**，**这个标记就是进位位** C，**在** PSW **中**。**在进行高位加法是将这个** C **加进去**。**例**：1067H+10A0H，**先做** 67H+A0H=107H，**而** 107H **显然超过了** 0FFH，**因此最终保存在** A **中的是** 7，**而** 1 **则到了** PSW **中的** CY **位了**，**换言之**，CY **就相当于是** 100H。**然后再做** 10H+10H+CY，**结果是** 21H，**所以最终的结果是** 2107H。

带借位的减法指令

SUBB A，Rn

SUBB A,direct

SUBB A,@Ri
SUBB A,#data
设（每个 H，（R2）=55H，CY=1，执行指令 SUBB A，R2 之后，A 中的值为 73H。
说明：没有不带借位的减法指令，如果需要做不带位的减法指令（在做第一次相减时），只要将 CY 清零即可。
乘法指令
MUL AB
此指令的功能是将 A 和 B 中的两个 8 位无符号数相乘，两数相乘结果一般比较大，因此最终结果用 1 个 16 位数来表达，其中高 8 位放在 B 中，低 8 位放在 A 中。在乘积大于 FFFFFH（65535）时，0V 置 1（溢出），否则 OV 为 0，而 CY 总是 0。
例：（A）=4EH，（B）=5DH，执行指令
MUL AB 后，乘积是 1C56H，所以在 B 中放的是 1CH，而 A 中放的则是 56H。
除法指令
DIV AB

此指令的功能是将 A 中的 8 位无符号数除了 B 中的 8 位无符号数（A/B）。除法一般会出现小数，但计算机中可没法直接表达小数，它用的是我们小学生还没接触到小数时用的商和余数的概念，如 13/5，其商是 2，余数是 3。除了以后，商放在 A 中，余数放在 B 中。CY 和 OV 都是 0。如果在做除法前 B 中的值是 00H，也就是除数为 0，那么 0V=1。
加 1 指令
INC A
INC Rn
INC direct
INC @Ri
INC DPTR

用途很简单，就是将后面目标中的值加 1。例：（A）=12H，（R0）=33H，（21H）=32H，（34H）=22H，DPTR=1234H。执行下面的指令：
INC A （A）=13H
INC R2 （R0）=34H
INC 21H （21H）=33H
INC @R0 （34H）=23H
INC DPTR （ DPTR）=1235H
后结果如上所示。

说明：从结果上看 INC A 和 ADD A，#1 差不多，但 INC A 是单字节，单周期指令，而 ADD #1 则是双字节，双周期指令，而且 INC A 不会影响 PSW 位，如（A）=0FFH，INC A 后（A）=00H，而 CY 依然保持不变。如果是 ADD A ，#1，则（A）=00H，而 CY 一定是 1。因此加 1 指令并不适合做加法，事实上它主要是用来做计数、地址增加等用途。另外，加法类指令都是以 A 为核心的&#0;&#0;其中一个数必须放在 A 中，而运算结果也必须放在 A 中，而加 1 类指令的对象则广泛得多，可以是寄存器、内存地址、间址寻址的地址等等。
减 1 指令
减 1 指令
DEC A
DEC RN
DEC direct
DEC @Ri

与加 1 指令类似，就不多说了。

综合练习：

```
MOV A，#12H
MOV R0，#24H
MOV 21H，#56H
ADD A，#12H
MOV DPTR，#4316H
ADD A，DPH
ADD A，R0
CLR C
SUBB A，DPL
SUBB A，#25H
INC A
SETB C
ADDC A，21H
INC R0
SUBB A，R0
MOV 24H，#16H
CLR C
ADD A，@R0
```

先写出每步运行结果，然后将以上题目建入，并在软件仿真中运行，观察寄存器及有关单元的内容的变化，是否与自已的预想结果相同。

单片机教程第十二课：逻辑运算类指令：

对累加器 A 的逻辑操作：
CLR A ；将 A 中的值清 0，单周期单字节指令，与 MOV A，#00H 效果相同。
CPL A ；将 A 中的值按位取反
RL A ；将 A 中的值逻辑左移
RLC A ；将 A 中的值加上进位位进行逻辑左移
RR A ；将 A 中的值进行逻辑右移
RRC A ；将 A 中的值加上进位位进行逻辑右移
SWAP A ；将 A 中的值高、低 4 位交换。
例：（A）=73H，则执行 CPL A，这样进行：
73H 化为二进制为 01110011，
逐位取反即为 10001100，也就是 8CH。
RL A 是将（A）中的值的第 7 位送到第 0 位，第 0 位送 1 位，依次类推。
例：A 中的值为 68H，执行 RL A。68H 化为二进制为 01101000，按上图进行移动。01101000 化为 11010000，即 D0H。
RLC A，是将（A）中的值带上进位位（C）进行移位。
例：A 中的值为 68H，C 中的值为 1，则执行 RLC A
1 01101000 后，结果是 0 11010001，也就是 C 进位位的值变成了 0，而（A）则变成了 D1H。
RR A 和 RRC A 就不多谈了，请大家参考上面两个例子自行练习吧。
SWAP A，是将 A 中的值的高、低 4 位进行交换。
例：（A）=39H，则执行 SWAP A 之后，A 中的值就是 93H。怎么正好是这么前后交换呢？因为这是一个 16 进制数，每 1 个 16 进位数字代表 4 个二进位。注意，如果是这样的：（A）=39，后面没 H，执行 SWAP A 之后，可不是（A）=93。要将它化成二进制再算：39 化为二进制是 10111，也就是 0001，0111 高 4 位是 0001，低 4 位是 0111，交换后是 01110001，也就是 71H，即 113。
练习，已知（A）=39H，执行下列指令后写出每步的结果
CPL A
RL A
CLR C
RRC A
SETB C
RLC A
SWAP A
通过前面的学习，我们已经掌握了相当一部份的指令，大家对这些枯燥的指令可能也有些厌烦了，下面让我们轻松一下，做个实验。
实验五：

```
ORG 0000H
LJMP START
ORG 30H
START:
```

```
MOV SP,#5FH
MOV A,#80H
LOOP:
MOV P1,A
RL A
LCALL DELAY
LJMP LOOP
delay:
mov r7,#255
d1: mov r6,#255
d2: nop
nop
nop
nop
djnz r6,d2
djnz r7,d1
ret
END
```

先让我们将程序写入片中，装进实验板，看一看现象。

看到的是一个暗点流动的现象，让我们来分析一下吧。

前而的 ORG 0000H、LJMP START、ORG 30H 等我们稍后分析。从 START 开始，MOV SP，#5FH，这是初始化堆栈，在本程序中有无此句无关紧要，不过我们慢慢开始接触正规的编程，我也就慢慢给大家培养习惯吧。

MOV A，#80H，将 80H 这个数送到 A 中去。干什么呢？不知道，往下看。

MOV P1，A。将 A 中的值送到 P1 端口去。此时 A 中的值是 80H，所以送出去的也就是 80H，因此 P1 口的值是 80H，也就是 10000000B，通过前面的分析，我们应当知道，此时 P1。7 接的 LED 是不亮的，而其它的 LED 都是亮的，所以就形成了一个“暗点”。继续看，RL A，RL A 是将 A 中的值进行左移，算一下，移之后的结果是什么？对了，是 01H，也就是 00000001B，这样，应当是接在 P1。0 上的 LED 不亮，而其它的都亮了，从现象上看“暗点”流到了后面。然后是调用延时程序，这个我们很熟悉了，让这个“暗点”“暗”一会儿。然后又调转到 LOOP 处（LJMP LOOP）。请大家计算一下，下面该哪个灯不亮了。。。。。对了，应当是接在 P1。1 上灯不亮了。这样依次循环，就形成了“暗点流动”这一现象。

问题：

如何实现亮点流动？

如何改变流动的方向？

答案：

1、将 A 中的初始值改为 7FH 即可。

2、将 RL A 改为 RR A 即可。

单片机教程第十三课：逻辑与指令

ANL A,Rn ;A 与 Rn 中的值按位'与'，结果送入 A 中
ANL A,direct ;A 与 direct 中的值按位'与'，结果送入 A 中
ANL A,@Ri ;A 与间址寻址单元@Ri 中的值按位'与'，结果送入 A 中
ANL A,#data ;A 与立即数 data 按位'与'，结果送入 A 中
ANL direct,A ;direct 中值与 A 中的值按位'与'，结果送入 direct 中
ANL direct,#data ;direct 中的值与立即数 data 按位'与'，结果送入 direct 中。
这几条指令的关键是知道什么是逻辑与。这里的逻辑与是指按位与
例：71H 和 56H 相与则将两数写成二进制形式：
（71H） 01110001
（56H） 00100110
结果 00100000 即 20H，从上面的式子可以看出，两个参与运算的值只要其中有一个位上是 0，则这位的结果就是 0，两个同是 1，结果才是 1。
理解了逻辑与的运算规则，结果自然就出来了。看每条指令后面的注释
下面再举一些例子来看。
MOV A，#45H ;(A)=45H
MOV R1，#25H ;(R1)=25H
MOV 25H，#79H ;(25H)=79H
ANL A，@R1 ;45H 与 79H 按位与，结果送入 A 中为 41H （A）=41H
ANL 25H,#15H ;25H 中的值（79H）与 15H 相与结果为（25H）=11H）
ANL 25H,A ;25H 中的值（11H）与 A 中的值（41H)相与，结果为(25H)=11H
在知道了逻辑与指令的功能后，逻辑或和逻辑异或的功能就很简单了。逻辑或是按位“或”，即有“1”为 1，全“0”为 0。例：
10011000
或 01100001
结果 11111001
而异或则是按位“异或”，相同为“0”，相异为“1”。例：
10011000
异或 01100001
结果 11111001
而所有的或指令，就是将与指仿中的 ANL 换成 ORL，而异或指令则是将 ANL 换成 XRL。即
或指令：
ORL A,Rn ;A 和 Rn 中的值按位'或'，结果送入 A 中
ORL A,direct ;A 和与间址寻址单元@Ri 中的值按位'或'，结果送入 A 中
ORL A,#data ;A 和立 direct 中的值按位'或'，结果送入 A 中
ORL A,@Ri ;A 和即数 data 按位'或'，结果送入 A 中
ORL direct,A ;direct 中值和 A 中的值按位'或'，结果送入 direct 中
ORL direct,#data ;direct 中的值和立即数 data 按位'或'，结果送入 direct 中。
异或指令：
XRL A,Rn ;A 和 Rn 中的值按位'异或'，结果送入 A 中
XRL A,direct ;A 和 direct 中的值按位'异或'，结果送入 A 中

XRL A,@Ri ;A 和间址寻址单元@Ri 中的值按位'异或'，结果送入 A 中
XRL A,#data ;A 和立即数 data 按位'异或'，结果送入 A 中
XRL direct,A ;direct 中值和 A 中的值按位'异或'，结果送入 direct 中
XRL direct,#data ;direct 中的值和立即数 data 按位'异或'，结果送入 direct 中。
练习：

```
MOV A，#24H
MOV R0，#37H
ORL A，R0
XRL A，#29H
MOV 35H，#10H
ORL 35H，#29H
MOV R0，#35H
ANL A，@R0
```

四、控制转移类指令
无条件转移类指令
短转移类指令
AJMP addr11
长转移类指令
LJMP addr16
相对转移指令
SJMP rel
上面的三条指令，如果要仔细分析的话，区别较大，但初学时，可不理会这么多，统统理解成：JMP 标号，也就是跳转到一个标号处。事实上，LJMP 标号，在前面的例程中我们已接触过，并且也知道如何来使用了。而 AJMP 和 SJMP 也是一样。那么他们的区别何在呢？在于跳转的范围不一样。好比跳远，LJMP 一下就能跳 64K 这么远（当然近了更没关系了）。而 AJMP 最多只能跳 2K 距离，而 SJMP 则最多只能跳 256 这么远。原则上，所有用 SJMP 或 AJMP 的地方都可以用 LJMP 来替代。因此在初学时，需要跳转时可以全用 LJMP，除了一个场合。什么场合呢？先了解一下 AJMP，AJMP 是一条双字节指令，也就说这条指令本身占用存储器（ROM）的两个单元。而 LJMP 则是三字节指令，即这条指令占用存储器（ROM）的三个单元。下面是第四条跳转指令。
间接转移指令
JMP @A+DPTR
这条指令的用途也是跳转，转到什么地方去呢？这可不能由标号简单地决定了。让我们从一个实际的例子入手吧。

```
MOV DPTR，#TAB ;将 TAB 所代表的地址送入 DPTR
MOV A，R0 ;从 R0 中取数（详见下面说明）
MOV B，#2
MUL A，B ;A 中的值乘 2（详见下面的说明）
JMP A，@A+DPTR ;跳转
TAB: AJMP S1 ;跳转表格
AJMP S2
AJMP S3
```

图 2
图 3

应用背景介绍：在单片机开发中，经常要用到键盘，见上面的 9 个按键的键盘。我们的要求是：当按下功能键 A……..G 时去完成不同的功能。这用程序设计的语言来表达的话，就是：按下不同的键去执行不同的程序段，以完成不同的功能。怎么样来实现呢？

看图 2，前面的程序读入的是按键的值，如按下'A'键后获得的键值是 0，按下'B'键后获得的值是'1'等等，然后根据不同的值进行跳转，如键值为 0 就转到 S1 执行，为 1 就转到 S2 执行。。。。如何来实现这一功能呢？

先从程序的下面看起，是若干个 AJMP 语句，这若干个 AJMP 语句最后在存储器中是这样存放的（见图 3），也就是每个 AJMP 语句都占用了两个存储器的空间，并且是连续存放的。而 AJMP S1 存放的地址是 TAB，到底 TAB 等于多少，我们不需要知道，把它留给汇编程序来算好了。

下面我们来看这段程序的执行过程：第一句 MOV DPTR，#TAB 执行完了之后，DPTR 中的值就是 TAB，第二句是 MOV A，R0，我们假设 R0 是由按键处理程序获得的键值，比如按下 A 键，R0 中的值是 0，按下 B 键，R0 中的值是 1，以此类推，现在我们假设按下的是 B 键，则执行完第二条指令后，A 中的值就是 1。并且按我们的分析，按下 B 后应当执行 S2 这段程序，让我们来看一看是否是这样呢？第三条、第四条指令是将 A 中的值乘 2，即执行完第 4 条指令后 A 中的值是 2。下面就执行 JMP @A+DPTR 了，现在 DPTR 中的值是 TAB，而 A+DPTR 后就是 TAB+2，因此，执行此句程序后，将会跳到 TAB+2 这个地址继续执行。看一看在 TAB+2 这个地址里面放的是什么？就是 AJMP S2 这条指令。因此，马上又执行 AJMP S2 指令，程序将跳到 S2 处往下执行，这与我们的要求相符合。

请大家自行分析按下键“A”、“C”、“D”……之后的情况。

这样我们用 JMP @A+DPTR 就实现了按下一键跳到相应的程序段去执行的这样一个要求。再问大家一个问题，为什么取得键值后要乘 2？如果例程下面的所有指令换成 LJMP，即：LJMP S1,LJMP S2……这段程序还能正确地执行吗？如果不能，应该怎么改？

单片机第十四课：条件转移指令

条件转移指令是指在满足一定条件时进行相对转移。

判 A **内容是否为** 0 **转移指令**

JZ rel

JNZ rel

第一指令的功能是：如果(A)=0，**则转移，否则顺序执行（执行本指令的下一条指令）。转移到什么地方去呢？如果按照传统的方法，就要算偏移量，很麻烦，好在现在我们可以借助于机器汇编了。因此这第指令我们可以这样理解：**JZ **标号。即转移到标号处。下面举一例说明：**

```
MOV A,R0
JZ L1
MOV R1,#00H
AJMP L2
L1: MOV R1,#0FFH
L2: SJMP L2
END
```

在执行上面这段程序前如果 R0 **中的值是** 0 **的话，就转移到** L1 **执行，因此最终的执行结果是** R1 **中的值为** 0FFH。**而如果** R0 **中的值不等于** 0，**则顺序执行，也就是执行** MOV R1，#00H **指令。最终的执行结果是** R1 **中的值等于** 0。

第一条指令的功能清楚了，第二条当然就好理解了，如果 A **中的值不等于** 0，**就转移。把上面的那个例子中的** JZ **改成** JNZ **试试吧，看看程序执行的结果是什么**?

比较转移指令

CJNE A,#data,rel

CJNE A,direct,rel

CJNE Rn,#data,rel

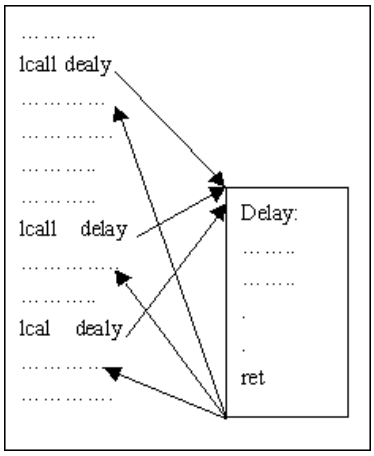
CJNE @Ri,#data,rel

第一条指令的功能是将 A **中的值和立即数** data **比较，如果两者相等，就顺序执行（执行本指令的下一条指令），如果不相等，就转移，同样地，我们可以将** rel **理解成标号，即：**CJNE A，#data,**标号。这样利用这条指令，我们就可以判断两数是否相等，这在很多场合是非常有用的。但有时还想得知两数比较之后哪个大，哪个小，本条指令也具有这样的功能，如果两数不相等，则** CPU **还会反映出哪个数大，哪个数小，这是用** CY**（进位位）来实现的。如果前面的数（**A **中的）大，则** CY=0，**否则** CY=1，**因此在程序转移后再次利用** CY **就可判断出** A **中的数比** data **大还是小了。**

例：

```
MOV A,R0
CJNE A,#10H,L1
MOV R1,#0FFH
AJMP L3
L1: JC L2
MOV R1,#0AAH
AJMP L3
```

L2: MOV R1,#0FFH
L3: SJMP L3
上面的程序中有一条指令我们还没学过，即 JC，这条指令的原型是 JC rel,作用和上面的 JZ 类似，但是它是判 CY 是 0，还是 1 进行转移，如果 CY=1，则转移到 JC 后面的标号处执行，如果 CY=0 则顺序执行（执行它的下面一条指令）。
分析一下上面的程序，如果（A）=10H，则顺序执行，即 R1=0。如果（A）不等于 10H，则转到 L1 处继续执行，在 L1 处，再次进行判断，如果（A）>10H，则 CY=1，将顺序执行，即执行 MOV R1，#0AAH 指令，而如果（A）<10H，则将转移到 L2 处指行，即执行 MOV R1，#0FFH 指令。因此最终结果是：本程序执行前，如果（R0）=10H，则（R1）=00H，如果（R0）>10H，则（R1）=0AAH，如果（R0）<10H，则（R1）=0FFH。
弄懂了这条指令，其它的几条就类似了，第二条是把 A 当中的值和直接地址中的值比较，第三条则是将直接地址中的值和立即数比较，第四条是将间址寻址得到的数和立即数比较，这里就不详谈了，下面给出几个相应的例子。
CJNE A,10H ;把 A 中的值和 10H 中的值比较（注意和上题的区别）
CJNE 10H，#35H ;把 10H 中的值和 35H 中的值比较
CJNE @R0,#35H ;把 R0 中的值作为地址，从此地址中取数并和 35H 比较
循环转移指令
DJNZ Rn,rel
DJNZ direct,rel
第一条指令在前面的例子中有详细的分析，这里就不多谈了。第二条指令，只是将 Rn 改成直接地址，其它一样，也不多说了，给一个例子。
DJNZ 10H，LOOP
3．调用与返回指令
（1）主程序与子程序 在前面的灯的实验中，我们已用到过了子程序，只是我们并没有明确地介绍。子程序是干什么用的，为什么要用子程序技术呢？举个例子，我们数据老师布置了 10 道算术题，经过观察，每一道题中都包含一个（3*5+2）*3 的运算，我们可以有两种选择，第一种，每做一道题，都把这个算式算一遍，第二种选择，我们可以先把这个结果算出来，也就是 51，放在一边，然后要用到这个算式时就将 51 代进去。这两种方法哪种更好呢？不必多言。设计程序时也是这样，有时一个功能会在程序的不同地方反复使用，我们就可以把这个功能做成一段程序，每次需要用到这个功能时就“调用”一下。
（2）调用及回过程：主程序调用了子程序，子程序执行完之后必须再回到主程序继续执行，不能“一去不回头”，那么回到什么地方呢？是回到调用子程序的下面一条指令继续执行（当然啦，要是还回到这条指令，不又要再调用子程序了吗？那可就没完没了了……）。参考图 1



调用指令

LCALL addr16 ;**长调用指令**

ACALL addr11 ;**短调用指令**

上面两条指令都是在主程序中调用子程序，两者有一定的区别，但在初学时，可以不加以区分，而且可以用 LCALL **标号，** ACALL **标号，来理解，即调用子程序**。

（5）**返回指令 则说了，子程序执行完后必须回到主程序，如何返回呢？只要执行一条返回指令就可以了，即执行** ret **指令**

4．**空操作指令**

nop **空操作，就是什么事也不干，停一个周期，一般用作短时间的延时**。

单片机第十五课：位及位操作指令

通过前面那些流水灯的例子，我们已经习惯了“位”一位就是一盏灯的亮和灭，而我们学的指令却全都是用“字节”来介绍的：字节的移动、加法、减法、逻辑运算、移位等等。用字节来处理一些数学问题，比如说：控制冰箱的温度、电视的音量等等很直观，可以直接用数值来表在。可是如果用它来控制一些开关的打开和合上，灯的亮和灭，就有些不直接了，记得我们上次课上的流水灯的例子吗？我们知道送往 P1 口的数值后并不能马上知道哪个灯亮和来灭，而是要化成二进制才知道。工业中有很多场合需要处理这类开关输出，继电器吸合，用字节来处理就显示有些麻烦，所以在 8031 单片机中特意引入一个位处理机制。

位寻址区

在 8031 中，有一部份 RAM 和一部份 SFR 是具有位寻址功能的，也就是说这些 RAM 的每一个位都有自已的地址，可以直接用这个地址来对此进行操作。

内部 RAM 的 20H-2FH 这 16 个字节，就是 8031 的位寻址区。看图 1。可见这里面的每一个 RAM 中的每个位我们都可能直接用位地址来找到它们，而不必用字节地址，然后再用逻辑指令的方式。

可以位寻址的特殊功能寄存器

8031 中有一些 SFR 是可以进行位寻址的，这些 SFR 的特点是其字节地址均可被 8 整除，如 A 累加器，B 寄存器、PSW、IP（中断优先级控制寄存器）、IE（中断允许控制寄存器）、SCON（串行口控制寄存器）、TCON（定时器/计数器控制寄存器）、P0-P3（I/O 端口锁存器）。以上的一些 SFR 我们还不熟，等我们讲解相关内容时再作详细解释。

位操作指令

MCS-51 单片机的硬件结构中，有一个位处理器（又称布尔处理器），它有一套位变量处理的指令集。在进行位处理时，CY（就是我们前面讲的进位位）称“位累加器”。有自已的位 RAM，也就是我们刚讲的内部 RAM 的 20H-2FH 这 16 个字节单元即 128 个位单元，还有自已的位 I/O 空间（即 P0.0…..P0.7,P1.0…….P1.7,P2.0……..P2.7,P3.0……..P3.7）。当然在物理实体上它们与原来的以字节寻址用的 RAM，及端口是完全相同的，或者说这些 RAM 及端口都可以有两种用法。

位传送指令

MOV C，BIT

MOV BIT，C

这组指令的功能是实现位累加器（CY）和其它位地址之间的数据传递。

例：MOV P1.0,CY ;将 CY 中的状态送到 P1.0 引脚上去（如果是做算术运算，我们就可以通过观察知道现在 CY 是多少啦）。

MOV P1.0,CY ;将 P1.0 的状态送给 CY。

位修正指令

位清 0 指令

CLR C ;使 CY=0

CLR bit ;使指令的位地址等于 0。例：CLR P1.0 ;即使 P1.0 变为 0

位置 1 指令

SETB C ;使 CY=1

SETB bit ;使指定的位地址等于 1。例：SETB P1.0 ;使 P.0 变为 1

位取反指令
CPL C ;使 CY 等于原来的相反的值，由 1 变为 0，由 0 变为 1。
CPL bit ;使指定的位的值等于原来相反的值，由 0 变为 1，由 1 变为 0。
例：CPL P1.0
以我们做过的实验为例，如果原来灯是亮的，则执行本指令后灯灭，反之原来灯是灭的，执行本指令后灯亮。
位逻辑运算指令
位与指令
ANL C,bit ;CY 与指定的位地址的值相与，结果送回 CY
ANL C,/bit ;先将指定的位地址中的值取出后取反，再和 CY 相与，结果送回 CY，但注意，指定的位地址中的值本身并不发生变化。
例：ANL C,/P1.0
设执行本指令前，CY=1，P1.0 等于 1（灯灭），则执行完本指令后 CY=0，而 P1.0 也是等于 1。
可用下列程序验证：
```
ORG 0000H
AJMP START
ORG 30H
START:  MOV SP，#5FH
MOV P1，#0FFH
SETB C
ANL C，/P1.0
MOV P1.1,C ;将做完的结果送 P1.1,结果应当是 P1.1 上的灯亮，而 P1.0 上的灯还是不亮
```
位或指令
```
ORL C,bit
ORL C,/bit
```
这个的功能大家自行分析吧，然后对照上面的例程，编一个验证程序，看看你相得对吗？
位条件转移指令
判 CY 转移指令
```
JC rel
JNC rel
```
第一条指令的功能是如果 CY 等于 1 就转移，如果不等于 1 就顺序执行。那么转移到什么地方去呢？我们可以这样理解：JC 标号，如果等于 1 就转到标号处执行。这条指令我们在上节课中已讲到，不再重复。
第二条指令则和第一条指令相反，即如果 CY=0 就转移，不等于 0 就顺序执行，当然，我们也同样理解： JNC 标号
判位变量转移指令
```
JB bit,rel
JNB bit,rel
```
第一条指令是如果指定的 bit 位中的值是 1，则转移，否则顺序执行。同样，我们可以这样理解这条指令：JB bit,标号
第二条指令请大家先自行分析
下面我们举个例子说明：
```
ORG 0000H
```

```
LJMP START
ORG 30H
START：MOV SP，#5FH
MOV P1，#0FFH
MOV P3，#0FFH
L1：JNB P3.2,L2 ;P3.2 上接有一只按键，它按下时，P3.2=0
JNB P3.3,L3 ;P3.3 上接有一只按键，它按下时，P3.3=0
LJM P L1
L2：MOV P1,#00H
LJMP L1
L3：MOV P1,#0FFH
LJMP L1
END
```

把上面的例子写入片子，看看有什么现象……

按下接在 P3.2 上的按键，P1 口的灯全亮了，松开或再按，灯并不熄灭，然后按下接在 P3.3 上的按键，灯就全灭了。这像什么？这不就是工业现场经常用到的“启动”、“停止”的功能吗？怎么做到的呢？一开始，将 0FFH 送入 P3 口，这样，P3 的所有引线都处于高电平，然后执行 L1，如果 P3.2 是高电平（键没有按下），则顺序执行 JNB P3.3,L3 语句，同样，如果 P3.3 是高电平（键没有按下），则顺序执行 LJMP L1 语句。这样就不停地检测 P3.2、P3.3，如果有一次 P3.2 上的按键按下去了，则转移到 L2，执行 MOV P1，#00H，使灯全亮，然后又转去 L1，再次循环，直到检测到 P3.3 为 0，则转 L3，执行 MOV P1，#0FFH，例灯全灭，再转去 L1，如此循环不已。

大家能否稍加改动，将本程序用 JB 指令改写？

伪指令

伪指令是对汇编起某种控制作用的特殊命令，其格式与通常的操作指令一样，并可加在汇编程序的任何地方，但它们并不产生机器指令。

许多伪指令要求带参数，这在定义伪指令时由“表达式”域指出，任何数值与表达式勻可以作为参数。

不同汇编程序允许的伪指令并不相同，以下所述的伪指令仅适用于ＭＡＳＭ５１系统，但一些基本的伪指令在大部份汇编程序中都能使用，当使用其它的汇编程序版本时，只要注意一下它们之间的区别就可以了。MASM51 中可用的伪指令有：

ORG 设置程序起始地址
END 标志源代码结束
EQU 定义常数
SET 定义整型数
DATA 给字节类型符号定值
BYTE 给字节类型符号定值
WROD 给字类型符号定值
BIT 给位地址取名
ALTNAME 用自定义名取代保留字
DB 给一块连续的存储区装载字节型数据
DW 给一块连续的存储区装载字型数据

DS 预留一个连续的存储区或装入指定字节。
INCLUDE 将一个源文件插入程序中
TITLE 列表文件中加入标题行
NOLIST 汇编时不产生列表文件
NOCODE 条件汇编时，条件为假的不产生清单
一、ORG

伪指令 ORG 用于为在它之后的程序设置地址值，它有一个参数，其格式为：
ORG 表达式

表达式可以是一个具体的数值，也可以包含变量名，如果包含变量名，则必须保证，当第一次遇到这条伪指令时，其中的变量必须已有定义（已有具体的数值），否则，无定义的值将由 0 替换，这将会造成错误。在列表文件中，由 ORG 定义的指令地址会被打印出来。

ORG 指令有什么用途呢？指令被翻译成机器码后，将被存入系统的ＲＯＭ中，一般情况下，机器码总是一个接一个地放在存储器中，但有一些代码，其位置有特殊要求，典型的是五个中断入口，它们必须被放在 0003H,000BH,0013H,001BH 和 0023H 的位置，否则就会出错，如果我们编程时不作特殊处理，让机器代码一个接一个地生成，不能保证这些代码正好处于这些规定的位置，执行就会出错，这时就要用到 ORG 伪指令了。看如下例子：
例：

```
    INT_0 EQU 1000H
    TIME_0 EQU 1010H
    INT_1 EQU 1020H
    TIME_1 EQU 1030H
    SERIAL EQU 1040H
    AJMP START ;跳转到主程序起始点
    LJMP INT_0 ;外中断 0 处理程序
    LJMP TIME_0 ;定时中断 0 处理程序
    LJMP INT_1 ;外中断 1 处理程序
    LJMP TIME_1 ;定时中断 1 处理程序
    LJMP SERIAL ;串行口中断程序
START:
    NOP
    END
```

上面的程序经汇编后列表文件如下:

The Cybernetic Micro Systems 8051 Family Assembler, Version 3.03 Page 1

08-26-96

```
1000 = INT_0 EQU 1000H

1010 = TIME_0 EQU 1010H
```

```
1020 = INT_1 EQU 1020H

1030 = TIME_1 EQU 1030H

1040 = SERIAL EQU 1040H

0000 0111 AJMP START ;跳转到主程序起始点

0002 021000 LJMP INT_0 ;外中断 0 处理程序

0005 021010 LJMP TIME_0 ;定时中断 0 处理程序

0008 021020 LJMP INT_1 ;外中断 1 处理程序

000B 021030 LJMP TIME_1 ;定时中断 1 处理程序

000E 021040 LJMP SERIAL ;串行口中断程序

START:

0011 00 NOP

0000 END
```



```
;%T Symbol Name Type Value

INT_0 . . . . . . . . . . . . . I 1000

INT_1 . . . . . . . . . . . . . I 1020

SERIAL. . . . . . . . . . . . . I 1040

START . . . . . . . . . . . . . L 0011

TIME_0. . . . . . . . . . . . . I 1010

TIME_1. . . . . . . . . . . . . I 1030
```

;%Z

00 Errors (0000)

由列表文件，可以绘出代码在ＲＯＭ中的映象图如下：

代码	01H	11H	02H	10H	00H	02H	10H	10H	02H	10H	20H
地址	00H	01H	02H	03H	04H	05H	06H	07H	08H	09H	0AH
代码	02H	10H	30H	02H	10H	40H	00H				
地址	0BH	0CH	0DH	0EH	0FH	10H	11H	12H	13H	14H	15

由上面的映象图可知,在０３Ｈ处的代码为１０Ｈ，而不是我们要的０２Ｈ，所以外断程序ＩＮＴ__0不能被正确执行，其它各中断程序的情况同样如此，如在０ＢＨ处，本来存放的应当是定时器0中断程序，但按上述的映象图，０ＢＨ处开始的3个代码是：０２Ｈ，１０Ｈ，３０Ｈ，这是定时器1的入口地址，所以，如果定时器0发生中断，所执行的其实是定时器1的中断程序，这当然不对。

例2：

```
INT_0 EQU 1000H

TIME_0 EQU 1010H

INT_1 EQU 1020H

TIME_1 EQU 1030H

SERIAL EQU 1040H

AJMP START ;跳转到主程序起始点

ORG 0003H

LJMP INT_0 ;外中断0处理程序

ORG 000BH

LJMP TIME_0 ;定时中断0处理程序
```

```
ORG 0013H

LJMP INT_1 ;外中断 1 处理程序

ORG 001BH

LJMP TIME_1 ;定时中断 1 处理程序

ORG 0023H

LJMP SERIAL ;串行口中断程序

START:

NOP

END
```

上面的程序经过汇编后列表文件如下：

```
The Cybernetic Micro Systems 8051 Family Assembler, Version 3.03 Page 1

08-26-96

1000 = INT_0 EQU 1000H

1010 = TIME_0 EQU 1010H

1020 = INT_1 EQU 1020H

1030 = TIME_1 EQU 1030H

1040 = SERIAL EQU 1040H

0000 0126 AJMP START ;跳转到主程序起始点

0003 ORG 0003H

0003 021000 LJMP INT_0 ;外中断 0 处理程序

000B ORG 000BH

000B 021010 LJMP TIME_0 ;定时中断 0 处理程序
```

0013 ORG 0013H

0013 021020 LJMP INT_1 ;**外中断** 1 **处理程序**

001B ORG 001BH

001B 021030 LJMP TIME_1 ;**定时中断** 1 **处理程序**

0023 ORG 0023H

0023 021040 LJMP SERIAL ;**串行口中断程序**

START:

0026 00 NOP

0000 END

The Cybernetic Micro Systems 8051 Family Assembler, Version 3.03 Page 2

08-26-96

;%T Symbol Name Type Value

INT_0 . . . . . I 1000

INT_1 . . . . . I 1020

SERIAL. . . . . I 1040

START . . . . . L 0026

TIME_0. . . . . I 1010

TIME_1. . . . . I 1030

;%Z

00 Errors (0000)

由列表文件，可以绘出代码在ＲＯＭ**中的映象图如下**：

代码	01H	11H		02H	10H	00H				
地址	00H	01H	02H	03H	04H	05H	06H	07H	08H	09H	0AH
代码	02H	10H	10H						02H	01H	20H
地址	0BH	0CH	0DH	0EH	0FH	10H	11H	12H	13H	14H	15H
代码						02H	10H	30H			
地址	16H	17H	18H	19H	1AH	1BH	1CH	1DH	1EH	1FH	20H
代码			02H	10H	40H	00H					
地址	21H	22H	23H	24H	25H	26H	27H	28H	29H	2AH	2BH

由映象图可知，各中断程序的代码都在其规定地址处，一旦产生中断即可执行相应的程序。至于图中未填的部分（如 02H)，**根据各编程器不同而不同，一般为** FFH **或** 00H。

二、END

END **语句标志源代码的结束，汇编程序遇到** END **语句即停止运行。若没有** END **语句，汇编将报错。**END **语句有一个参数，可以是数值** 0，**也可以是表达式，其格式是：**

标号: END **表达式**

它的值就是程序的地址并且作为一个特殊的记录写入 HEX **文件。若这个表达式省略，**HEX **文件中其值就是** 0。

三、EQU

EQU **以及其它一些符号定义伪指令用来给程序中出现的一些符号赋值。对这些符号名的要求与其它符号相同，即长度不限，大小写字母可互换并且必须以字母开头。**

由等值指令定义的符号是汇编符号表的一部分。等值伪指令有两种形式。一种用 EQU，**另一种用字符“**=”**即**

符号名 EQU **表达式**

符号名 = **表达式**

两种形式的效果是一样的。符号名在左边，其对应的值在右边。值可以是变元，其它的符号名或表达式。只要在两遍扫描中求出表达式的值就行，否则引用该符号名时将报错。当表达式的值是字符串时，只取后两个字符。若串长为 1，**高位字节被置** 0，**符号名的值被打印在程序清单中。由等值伪指令定义的符号名不允许重名。如果经定义的符号名被重定义，则汇**

编将报出错，并且这个符号名按新定义的处理，最好不要在程序中出现重名。

例：0469= ABC EQU 469H

0464= XY EQU ABC-5

02F0= JK = 752

0754 XYJK = XY+JK

在列表文件中最左边的数字不是这些伪指令所在的地址而是通过汇编后赋给符号名的值。第一条符号名 ABC **被起来** 469H，**第二条** XY **被赋于** ABC-5，**因此** XY **的值为** 469H-5=464H，JK **的值为** 752（**即** 2F0H)，XYJK **的值** XY+JK=464H+2F0H=754H

四、SET

SET **伪指令有些类似于等值伪指令，它定义了一个整数类型的符号名，它的格式为**

符号名 SET **表达式**

SET **伪指令与等值伪指令的唯一区别在于** SET **伪指令所定义的符号名右以在程序中多次定义，而不报错**。

例：002D= K57 SET 101101B

8707= K57 SET 34567

五、DATA **与** BYTE

DATA **与** BYTE **都是用来定义字节类型的存储单元，赋予字节类型的存储单元一个符号名，以便在程序中通过符号名来访问这个存储单元，以帮助对程序的理解**。

BYTE **与** DATE **之间的区别类似于** EQU **和** SET，BYTE **伪指令不能定义重名**。

六、WORD

WORD **伪指令类似于** DATE **伪指令，只是** WORD **伪指令定义了一个字类型的符号名，其格式为**：

符号名 WORD **表达式**

0027= VAL31 WORD 39

0021= PAR7 WORD 21H

一个字由 2 **个字节组成。当然，因为** ８０５１**汇编语言集没有字操作，所以程序执行时，只处理字节。**ＷＲＯＤ**伪指令仅仅允许用户定义一个认为是字的存储位置。**

七、BIT

BIT **伪指令定义了一个字位类型的符号名，其格式为：**

符号名 BIT **表达式**

这里表达式的值是一个位地址，这个伪指令有助于位的地址符号化。

例：

002F= LOG3 BIT 47

0014= Y731 BIT 14H

八、ALTNAME

替换名（ALTNAME**）伪指令提供用户一种手段，以定义一个符号名来替换一个保留字，此后这个答名与被替换的保留字均可等效地用于程序中。任何保留类型的答名均可被替换。替换名伪指令格式为：**

ALTNAME **保留字，新名**

例：

0002= ALTNAME R2 COUNT

013A EA MOV A,R2

013B E502 MOV A,COUNT

九、ＤＢ

ＤＢ**伪指令用于定义一个连续的存储区，给该存储区的存储单元赋值。该伪指令的参数即为存储单元的值，在表达式中对变元个数没有限制，只要此条伪指令能容纳在源程序的一行内，其格式为：**

标号： DB 表达式

只要表达式不是字符串，每一表达式值都被赋给一个字节。计算表达式值时按１６位处理，但其结果只取低８位，若多个表达式出现在一个ＤＢ伪指令中，它们必须以逗号分开。

表达式中有字符串时，以单引号“”作分隔符，每个字符占一个字节，字符串不加改变地被存在各字节中，并不将小写字母转换成大写字母。

例如：

DB 00H 01H 03H 46H

DB 'This is a demo!'

十、DW

DW **为以字节为单元（十六位二进制）来给一个的存储区赋值，其格式为：**

标号： DW 表达式

例如：

0000 3035 D46B DW 12341,54379,10110100101110B

0004 2D2E

0006 4344 4243 DW 'ABCD','BC','A'

000A 0041

000C 2868 02E8 DW 456*375h,83+295h,'YZ',72h-456

0010 595A FEAA

十一、DS

DS **为定义存储内容的伪指令，用它定义一个存储区，并用指定的参数填满该存储区。**

DS **伪指令包含两个变元，第一个变元定义了存储区的长度的字节数，在汇编时，汇编程序将跳过这些单元把其它指令汇编在这些字节之后，因此在使用ＤＳ伪指令时第一个变元不可活力第二个变元表示在这些单元中真入什么值，第二个变元可以活力活力时这些字节将不处**

理。下例中 0173 处有一条 DS 9，则空出 9 个字节，下一第指令被汇编到 017C 处；在 017C 处空出 1BH 个单元，在这些字节中被 27H 所填充。

DS 指令的格式如下：

标号： DS 表达式 1，表达式 2

表达式 1 定义了存储区的长度（以字节为单位）。这个变元不能省略。表达式 2 是可选择的，它的值低 8 位用以填入所定义的存储区。

若省略则这部分存储单元不处理。

例：

0000 04 INC A

0001 DS 9

000A 04 INC A

000B DS 1BH,27H

0026 04 INC A