51单片机汇编语言教程：第10课-数据传送类指令

（基于 HJ-1G、HJ-3G 实验板）

单片机的累加器 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系列单片机中，与外部存储器 RAM 打交道的只能是 A 累加器。所有需要传送入外部 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。在此我们能看出内外部 RAM 的区别了，内部 RAM 间能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，然后再传送到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，在后两条单片机指令中，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，所以只供给低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，就只需要供给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：将单片机外部 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。本指令也被称为单片机查表指令，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办法：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，要求用查表的办法确定它的平方值（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。其它数据也能类推。

标号的真实含义：从这个地方也能看到另一个问题，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 单片机指令中，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。

能通过以下的例程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。这个能看懂了吧？

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&#0;&#0;指我们用的电脑去做了。

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 指令的在单片机中执行是这样的，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 后一般要执行其他指令，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，单片机也是一样,而我们在编写子程序，需要保存数据时，常常也不采用后面的办法，而是用堆栈的办法来实现。

例：写出以下单片机程序的运行结果

```
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。从这个例程能看出：使用堆栈时，入栈的书写次序和出栈的书写次序必须相反，才能保证数据被送回原位，不然就要出错了。