DJS—130 符号程序设计基础

北京钢铁学院
计算机应用教研室翻印
一九七八年十月

目　录

前　言

DJS-130系列机的字长为16位：从左到右为0至15位，磁心存贮器的最大容量为32K（32768）个字。因此，存贮器的寻址范围从：

0至32767（10）

或0至77777（8）

而且用一个15位的字就能够寻址存贮器的全部单元。因而程序计数器长为15位，在存贮器中存贮地址占据1至15的位置。

计算机有四个16位的累加器（AC_0、AC_1、AC_2、AC_3），在其中完成全部算术逻辑操作并通过它们完成所有的输入输出传送。

与累加器组合在一起的是进位标志，它指示由ϕ位产生的算术进位

计算机有一个15位的程序计数器（PC），它寄存下一条指令存贮单元的地址。每一条指令执行后，程序计数器加1，于是产生顺序程序流。正常的程序流可用一条SKIP或JUMP指令改变。

在计算机的控制台上有几个开关，用这些开关可以手动向存贮器和累加器输入数据，并且可以控制程序和计算机操作。还有一些指示灯等。

DJS-130系列计算机的指令可分成三类：

访问内存指令：这类指令包括有在累加器和存贮器之间传送数据的指令，修改存贮器的指令，和改变程序流的转移指令。

算术和逻辑指令：这类指令包括有：处理累加器内容和进位标志的指令，以及在累加器之间完成所有的算术和逻辑操作的指令。

输入输出指令：这类指令包括有：在累加器和入出外部装置之间传送数据的指令和只用来控制入出装置的指令。

第一章　DJS—130机指令系统

程序的编制工作，主要涉及这样两个问题：第一是指令系统的问题。第二是编制步骤的问题。所谓“指令系统”，是指某一台具体的电子计算机的指令系统。不同的电子计算机，它所执行的全部指令，称为这台电子计算机的指令系统。不同的电子计算机有着不同的指令系统。我们编制某一个算题程序，总是针对某一台计算机的，因此必须首先熟悉这台计算机的指令系统。第二是怎样针对要解的数学问题，编制出算题程序。一般说来可分为：确定算法，画出程序框图，编制符号程序、分配内存、进行代真等几个基本步骤。下边分别讲述。

第一节　訪內指令

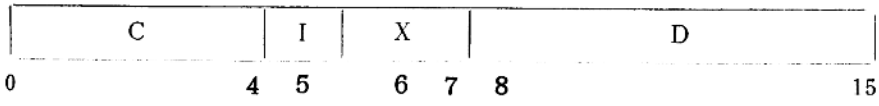
每个16位的访内指令字分成四段：

指令段（C）　　ϕ～4位

寻址方式段（I）　　5位

变址段（X）　　6～7位

相对地址段（D）　　8～15位



指令段规定指令的类型：传送数据；修改存贮器；转移。

每条访内指令必须包括有一个有效地址（E），它确定访问的存贮单元。有效地址（E）由变址（X）和相对地址D形成。变址X是指某个寄存器或累加器，它的内容加上相对地址D，产生所要求的存贮单元的地址。

$$E=(x)+D$$

此处（ ）的意思是指变址器中的内容。

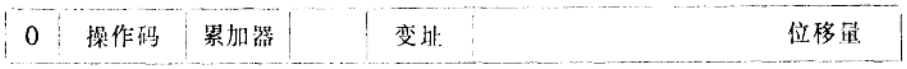
如果变址X是O，则D是不带符号的，其范围可以是000至377(8)，变址X是1，2或3时，相对地址D是带符号的数，其值为：

正值：000～177(8)　　或　0～127(10)

负值：377～200(8)　　－1～－128(10)

换句话说，如果8位（D的最左边的一位）是O，D则是一个正值，如果8位是1，则D表示为一个负的相对地址。为求得有效地址，把D加到由变址X所确定的寄存器或累加器的内容上。

一、用累加器的访内指令



0 1 2 3 4 5 6 7 8 15

→ 地址类型

0 1 L D A

1 0 S T A

当累加器部分为：

0 0 代表AC0

0 1 代表AC 1

1 0 代表AC 2

1 1 代表AC 3

变址部分 X：

0 0表示取0页地址。这时位移量D是一个无符号数，取值范围是0～377。所以它相当于指令的编址区域是00000～00377。

0 1表示相对地址修饰。这时位移量D是个有符号数，第8位为符号位，取值范围是－200－177。这时的有效地址：

$$E=(PC)+D$$

1 0表示变址器修饰，变址器为AC2，D仍是个有符号数，$-200\leq D\leq 177$。这时的有效地址：

$$E=(AC_2)+D$$

11也是变址器修饰，变址器为AC_3，D也仍是个有符号的数，$-200\leq D\leq 177$。这时有效地址为：

$$E=(AC_3)+D$$

综合起来可以列表如下：

X	变址方法	D	有效地址 d	备　注
0	0页地址	000～377	E＝D	X＝0可以不写明
1	相对变址	－200～177	E＝(PC)＋D	可缩写为·±(D)
2	以AC_2为变址器	－200～177	E＝(AC_2)＋D	
3	以AC_3为变址器	－200～177	E＝(AC_2)＋D	

地址类型部分I：

0　代表直接地址；它的有效地址直接由X和D来确定

1　代表间接地址，这时要由X和D所确定的地址内容作为新的地址，在这个新地址中，第0位为间接标志位，第1～15位就是有效地址A

0 1 15

如果第0位是0，则A就是最后取得的有效地址，如果第0位是1，则再要从A中取其内容为地址，重复上面这个过程，直到第0位为0时为止，得到最后确定的有效地址E。

还有一点十分重要的是：在取间接地址的过程中，当取到00020～00037时，机器将把它们的内容自动加1或减1，然后送回到内存中去。再看第0位是0还是1、确定是以此地址为有效地址还是再间接取下一地址。当地址是00020～00027时是自动加1，00030～00037时是自动减1。这种功能在实际运用上是很有意义的。

LDA　　AC,　　D,　　X

这是一条由有效地址E所确定的存储单元的内容送入累加器AC的指令，称为取数指令。指令执行后，内存E的内容不变。

AC可以等于0，1，2，3。即指累加器AC0，AC1，AC2，AC3。

D可以等于 $-200 \leq D \leq 177$（当X＝1，2，3时）

或等于　$0 \leq D \leq 377$（当X＝0或无时）

X可以等于无，0，1，2，3。而0和无具有同样的效果。

在一般情况下，实际参加运算的有效地址，都要经过地址修饰而成，具体的说，就是要经过变址器修饰，相对地址修饰或间接地址修饰而成。

例1　假设下述地址的内容如下表所示：

地　址	内　容
6	100015
12	000035
15	000017
17	000023
23	000011
AC3	000015

下面我们就来举例说明这条指令的用法：

LDA　1,6表示把地址6的内容100015取到AC1，因此这条指令没有进行地址修饰。

LDA　1，－7，3这时，有效地址E＝(AC3)－7＝15－7＝6所以它仍表示100015⟶AC1。

LDA　1，@15这条指令要进行间接地址修饰，这时指令按间接地址15找到17，∴E＝17，指令就表示(17)＝23⟶AC1。

LDA　1，@　6　这条指令也要进行间接地址修饰，先由地址6找到100015，由于它的0位为1，所以再按间接地址15找到17，∴E＝17，指令仍表示(17)＝23⟶AC1。

LDA　1，6，3　这时，E＝(AC3)＋6＝15＋6＝23，即以这条指令表示(23)＝11⟶AC1

LDA　1，@23由于@23是在20～27之间，∴E＝11＋1＝12，它表示(12)＝35⟶AC1。

LDA　1，@6，3由于@6＋(AC3)＝@23，所所这条指令也表示把35⟶AC1。

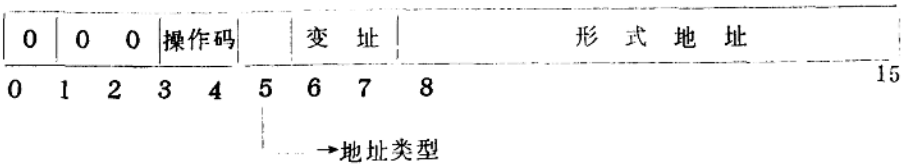
LDA　1，23　即把(23)＝11⟶AC1。

第二条指令STA（即STore Accumulator的缩写），也可以简写为：

STA　　AC,　　D,　　X

这是一条由累加器AC的内容向存储单元E传送的指令，在传送过程中，AC的内容不受影响，而存储单元E中原来的内容消失了。它的使用方法完全类似于取数指令LDA，这里就不再重复说明了。

二、不用累加的访内指令



操作码：

0 0	JMP
0 1	JSR
1 0	ISZ
1 1	DSZ

它们可以分成两组

（I）修改内存指令

这一部分指令有两条，用来改变内存单元的值並测定其结果是否跳跃、一般是作为计数时用的。

其符号指令格式为：

操作码	〔间接地址〕	地 址	〔变 址〕
JSZ		0	{0 1}
	@	∫	2
DSZ		377(8)	3

1、ISZ(Increment and Skip if resutl is Zero)

称为加“1”判另跳。可简写为

ISZ D, X

它是将有效地址E的内容加1並放回到E中去，判断结果是否等于0，是0则跳过一条指令执行，否则顺序作下一条指令。

2、DSZ (Decrement and Skip if result is Zero)称为减“1”判另跳。可简写为：

DSZ D, X

它是将有效地址E的内容减1並放回到E中去，判断结果是否等于0，是0则跳过一条指令执行，否则顺序作下一条指令。

（I）转移指令

这也有两条指令，用以改变程序当前执行的流程，是十分重要的指令。

其汇编格式为：

操作码	〔间接地址〕	地 址	〔变 址〕
JMP		0	{0 1}
	@	∫	2
JSR		377(8)	3

1、JMP (Jump) 称为无条件转移。可简写为

JMP D, X

它将有效地址E送到PC中去，无条件地使程序改变现有的执行顺序並从E处开始执行下面的指令。

例2 将2000～2035单元中的内容送到5150～5205中去，並使它们的次序颠倒。

解：

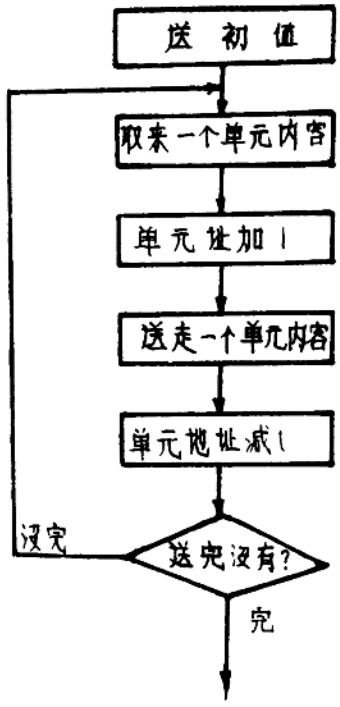


图 1

从框图中可以看出，每取一个数，地址要加 1，每送一个数的地址要减 1，因此编制这一段程序时应该利用自动变址的功能。

另外，初值包括三个。

① 2000～2035的初址，由于用了自动变址，所以改为1777；

② 5150～5205的终址，也由于了自动变址，改为5206；

③ 计数值 36(8)=30(10)

因此其程序为：

```
CNT: 001777:原数组初始地址减 1
     005206:现数组终止加 1          } 建立初期
     000036:计数值
     LDA  0,   CNT  ;
     STA  0,   21   ;               } 送初值
     LDA  0,   CNT+1;
     STA  0,   35   ;
```

```
LOOP: LDA  O, @  21     ; 取一个单元的内容
      STA  O, @  35     ; 送走一个单元的内容
      DSZ  CNT+2        ; 计数值减1
      JMP  LOOP         ; 没送完转向LOOP，再继续送
      ⋮    ⋮            ; 送完了，往下执行
```

2、JSR(Jump to Sub Routine)，称为转子程序。也可简写为

JSR　　D,　　X

其中D为形式地址，X为变址器。

在说明这条指令的功能以前，先介绍一下子程序的概念。

我们知道，计算机算题的内容各不相同，控制生产的问题也千差万别，由此编出的程序自然也不同。有的复杂，有的简单，但是任何复杂的问题都是由一些简单的，带有共同性的部分组合而成。程序也毫不例外。

例如我们平时算题常常要用到$\sqrt{X}$，SinX，e^X等初等函数以及10→2，2→10的数制转换；定点机上欲进行浮点运算、双字长运算；控制生产用的报表打印；……等等，举不胜举。就是某个算题的本身也常有一些公用部分。这些工作都不是用一条或几条指令而是要用一系列的指令才能完成。如果我们每作这些工作都要重新编制程序，便必影响算题任务的完成和浪费大量劳动。因此我们有必要把这些公用部分的程序独立成一个程序，它可以不和整个程序放在一起，使用时控制转向它，而作完后，又让它再返回原程序。这种程序就叫子程序，而调用子程序的程序叫做主程序。

因此为了能用子程序，必须要有这三种功能：①能转向它②能返回来③为了能返回来，必须记住返回地址，转子程序指令就能协助完成这些任务。

现在我们就来介绍这条指令。

JSR　　D,　　X

这条指令将PC内容加1的数（即返回地址）送到AC3，使AC3接受JSR的下一条指令地址；並让有效地址E送到PC，程序接着从E处开始执行下面的指令（即完成转向子程序的任务）。

有了这条指令，调用子程序就非常方便了，由于它能将PC＋1保留在AC3中，因此子程序的返回，只要在它的末尾安排一条指令

JMP　　O,　　3

现图示如下：

在这里需要注意：在JSR指令中，有效地址的计算是先于PC＋1送AC3的，因此JSR指令仍可以用AC3作为变址寄存器，即JSR的有效地址的计算，仍可以用AC3原来的内容来计算。

例3　某个程序在3000(8)号单元要进行例2那样成组数据的传送，试写出其主程序和子程序。

解：成组数据传送是个经常要用到的功能，我们可以把它安排为子程序，因此其程序为：

```
主　程　序
      ⋮
      ⋮
```

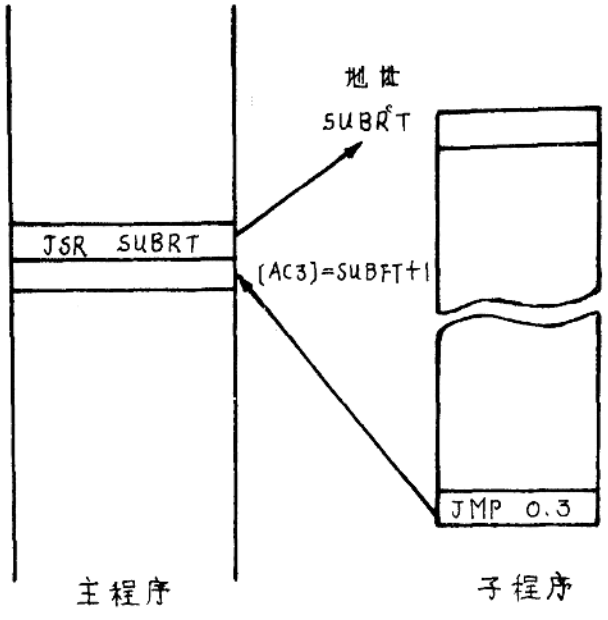


图2

```
3000  JSR   .DATA;   转向子程序
        ⋮
子      程      序
CNT:  001777;  传出的首地址
      005206;  传向的首地址
      000036;  成组数据的个数
.DATA;   LDA   0,   CNT
         STA   0,   21
         LDA   0,   CNT+1
         STA   0,   35
LOOP:    LDA   0,  @21
         STA   0,  @35
         DSZ   CNT+2
         JMP   LOOP
         JMP   0,   3;  完成返回的任务
```

但是这样安排的子程序有缺点，它的参数是固定的（参数是指1777，5206和36），因而缺乏通用性。在实际应用上，成组数据的传送並不一定限止在2000～2035⟶5150～5205。我们编写的子程序总是希望它带有通用性，参数是可以任意填写的，为此我们可以把程序改写成

```
主    程    序
          ⋮
3000    JSR   ·DATA
        参数1；例如可以是1777，也可以是其它数
        参数2；例如可以是520€，也可以是其它数
        参数3；例如可以是36，也可以是其它数
          ⋮
子    程    序
.DATA: LDA   0，0，3；取参数1 ⟶ AC0，∵(AC3)=3001
       LDA   1，1，3；取参数2 ⟶ AC1，∵(AC3)=3001
       LDA   2，2，3；取参数3 ⟶ AC2，∵(AC3)=3001
       STA   0，21
       STA   1，35          送初值
       STA   2，GSZ
LOOP: LDA   0，@ 21
       STA   0，@ 35
       DSZ   GSZ
       JMP   LOOP
       JMP   3，3；返回主程序
GSZ:                  成组数据个数暂存单元
```

如果给出的不是参数本身，而是参数的地址。那可以用间接地址的办法来解决。这时

```
主    程    序
          ⋮
        JSR   ·DATA
        参数1的地址
        参数2的地址
        参数3的地址
          ⋮
子    程    序
.DATA: LDA   0，@ 0，3；取参数1
       LDA   1，@ 1，3；取参数2
       LDA   2，@ 2，3；取参数3
          ⋮
       JMP   3，3；返回主程序
```

这样安排的子程序还有一个缺点：那就是要破坏原主程序累加器ACO，AC1，AC2的内容。有的时候主程序是不允许子程序破坏累加器内容的，为了解决这个问题，在子程序里就要给ACO，AC1，AC2的内容进行退避。

假设TBO，TB1，TB2是ACO，AC1，AC2内容的退避单元，那么子程序可编制成

```
DATA  STA  0，TBO；ACO的退避
      STA  1，TB1；AC1的退避
      STA  2，TB2；AC2的退避
      LDA  0，@0，3
      LDA  1，@1，3
      LDA  2，@2，3
       ⋮
      LDA  0，TBO；恢复  ACO
      LDA  1，TB1；恢复  ACO
      LDA  2，TB2；恢复  AC1
      JMP  3，3；恢复AC2
TBO：0；ACO的退避单元
TB1：0；AC1的退避单元
TB2：0；AC2的退避单元
```

从上述的例子可以看到，在编子程序的过程中，正确理解AC3的内容是很重要的，取参数和返回主程序都是靠它来完成的，在一般情况下它的内容是不准被破坏的，但是我们只有四个运算累加器，有的时候子程序本身还要用到AC3，为了保证子程序能正确地返回主程序，AC3也要进行退避。现举例如下：

例4 试编制计算双字长数相加的子程序，並规定被加数存放在ACO，AC1调用这一段程序的指令格式为：

```
JSR   .DADD
      加数高位地址
```

解：关于双字长加法的程序在此不具体进行讨论，现在的问题是要把它配成子程序。

在配这段子程序时，由于是双字长运算，所以四个累加器都要用，必须对AC3进行退避。其程序为：

```
.DADD：STA  3，BDO3；AC3的退避
       ISZ .BDO3      ；因为调用这个子程
                      ；序时，指令格式中有一
                      ；个参数，所以返回时要
                      ；加1
       LDA  3，0，3；加数高位的地址→AC3
       LDA  2，0，3；加数的高位数→AC2
       LDA  3，1，3；加数的低位数→AC3
       ADDZ3，1，SZC
       INC  0，0
```

```
        ADD 2，0
        JMP @.BDO3；返回
BDO3：  0                ；保存返回的单元
```

在这段程序里，主程序的参数虽然给的是高位数的地址，但我们没有用间接地址，而是多用了一条LDA3，0，3指令。这自然也是可以的。

编制子程序的技术是很重要的，根据使用上的不同要求，变化也是较多的，需要我们在实践中正确的掌握它，以便我们更好地为社会主义服务。

第二节　算术和邏輯指令

1	源累加器	目的累加器	操作码	移位	进位	开关	跳跃
0	1 2	3 4	5 6	7 8 9	10 11	12	13 14 15
		000	COM	00 ～	00 ～		000 ～
		001	NEG	01 L	01 Z		001 SKP
		010	MOV	10 R	10 O		010 SZC
		011	INC	11 S	11 C		011 SNC
		100	ADC				100 SZR
		101	SUB				101 SNR
		110	ADD				110 SEZ
		111	AND				111 SBN

这一大类指令是在二个累加器中进行的，一个称为源累加器（Source AC）ACS，一个称为目的累加器（Destination AC）ACD。

算术和逻辑指令的最大特点在于操作码部分，它较充分地发挥了指令各位的功能。在一条指令中，除了普通的操作码以外，称为基本操作，还安排了辅助操作，它包括移位、进位、输入开关和跳跃等操作。整个指令的功能是由这两种操作组合而成，这使指令的功能十分灵活，有利于提高程序的质量。

基本操作加上辅助操作总共有五种。它们在执行过程中有个严格的操作顺序，次序搞错了，对指令的理解上就会有问题，必须注意。它们的操作顺序如下图所示：

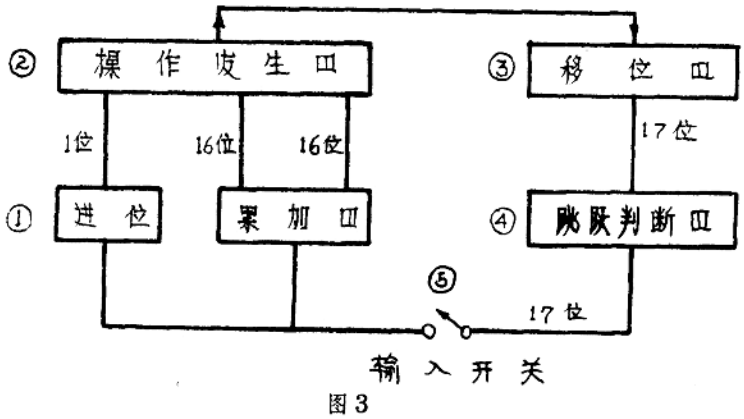


图3

其指令格式可统一写成：

操作码	进位	移位	输入开关	源累加器	目的累加器	跳跃
COM NEG MOV INC ADC SUB ADD AND	~ Z O C	~ R L S	[#]	ACS	ACD	~ SKP SZC SNC SZR SNR SEZ SBN

上面我们就逐个讨论这类指令的各种操作，並随后举一些例子。

（I）辅助操作

为了便于说明问题，我们先介绍辅助操作，然后和基本操作结合起来说明它们的功能。

1、进位（CARRY）

它是用来指出运算器在执行基本操作之前，提供给进位标志位的值的，称为进位标志位的初值，其作用如下：

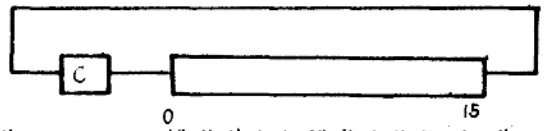
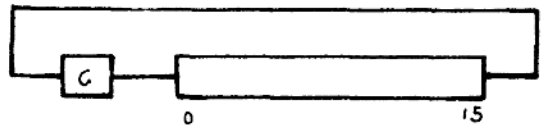
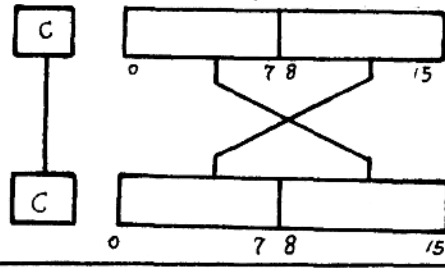
记忆码	移位操作
~	不移位
L（Left）	左移一位，0位移到进位位上去，进位位移到15位。 
R（Right	右移一位，15位移到进位位上去，进位位移移到0位。 
S（Swab）	把这林结果分成8位进行交换进位位不受影响。 

图 4

记忆码　进位标志位的初值

～　　取当前进位标志位的值送操作发生器

Z(Zero)把进位标志位按“0”送操作发生器

O(one) 把进位标志位按“1”送操作发生器

(Complement)把当前进位标志位的值取反后送操作发生器

2、移位 (Slift)

它在操作发生器执行完了以后把结果按图4规定进行移位。

3、跳跃 (Skip)

跳跃的功能总共有8种，它是在进行了移位操作以后，检验运算结果是否满足跳跃条件，如果满足，就跳过一条指令执行，不满足就继续顺序执行。具体的就是：

①～ (不写)　　不跳跃，空操作

②SKP(SKiP)　　无条件跳跃

③SZC(Skip on Zero Carry)

进位位为0时跳

④SNC(Skip on Non-zero Carry)

进位位不为0时跳

⑤SZR(Skip on Zero Result)

结果为0时跳

⑥SNR(Skip on Non-zero Result)

结果不为0时跳

⑦SEZ(Skip if Eitner carry or resuet is Zero)

结果或进位有一个为0时跳

⑧SBN(Skip if BOth carry and resuet Non-zero)

结果和进位位都不为0时跳

4、输入开关 (no load)

当操作码（指基本操作）的功能已经完成，移位也完成，並做了跳跃的判断，对于结果是否存入目的累加器ACD，就由这一项辅助操作来完成。

记忆码	结果是否存入ACD
～(不写)	存入ACD中
#	不存入ACD中，ACS和ACD的内容保持不变。

（I）基本操作

这一部分操作码也有8种，是算术逻辑运算指令的主干部分，它与辅助操作结合在一起，完成指令的操作功能，具体的有

1、取反　COM(COMplement)

将源累加器ACS的内容取反並把进位位C所给定的进位初值送到移位器，完成了这部分操作后，还能进行移位、跳跃和是否输入等辅助操作。

在不出现“#”号的条件下，完成

$(ACS)_{反} \rightarrow ACD$

例5 试判断AC1的内容是否是全“1”，是就跳过一条指令。

解：由于对判断AC1的内容是否是全“1”，没有直接的操作码可以利用，但对于判断是否为“0”是有的，即SZR。因此我们把COM和SZR配合起来就可以完成这个例子的任务，即采用

COM#1，3，SZR

在执行这条指令时，由于出现有#，AC1取反后的结果不放入AC_3，所以这条指令只起个判断的作用，並不改变任何累加器的内容。

2、取补NEG (NEGate)

这种操作码的功能是将源累器ACS的内容取补后送目的累加器（如不出现#），完成了这部分操作后，还能进行其他的辅助操作。所以在不出现“#”号的条件下，完成

$$-(ACS)\rightarrow ACD$$

3、传送MOV (MOVe)

将源累加器ACS的内容和进位位C指定的进位初值传送到移位器中去，並依次完成其它辅助操作。

在不出现“#”号的条件下，它将运算结果传送到ACD中去，即

$$(ACS)\longrightarrow ACD$$

例6 试编取ACO的内容的绝对值的程序

解： ACO的内容正也可负，所以我们先要判断一下它的符号，即第0位是0（正）还是1（负）。如果是正就不必处理它，如果是负，那就再取个负，负负得正。

在编程序时，对这样一段编程序的思想，习惯上用一个框图来表示，使其思想既明确又直观。对于这样一种方法，初接触时似乎显得不必要，但是以后逐渐编制一些复杂的程序，就会感到非常必要了。

现在我们把上面那段编程序的思想画成框图如下：

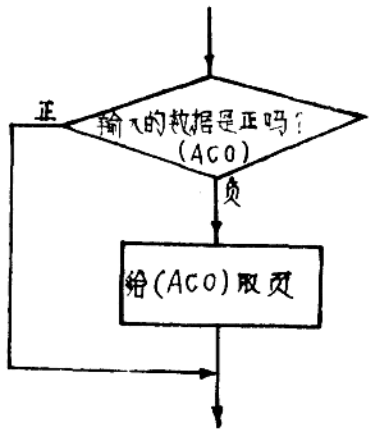


图5

在框图中习惯上把计算部分用方框表示，判别部分用棱形或椭圆形表示。程序就按一个框一个框地编制。

现编制程序如下：

MOVL# 0，0，SZC；检查符号，是正还是负，是正就跳

NEG 0，0 ；是负就再取个负

例7 试编制AC2的内容除2的结果的程序。

解： AC2的内容可以是正数也可以是负数，因此我们首先要判断它的第0位是0还是1。

为了把AC2的内容除2，那么只要把它们向右移一位就可以了。

但是移位后，正数要在左边补个0；负数就必须补个1（因负数是以补码形式出现的）。

把这两段编制程序的思想画成框图即

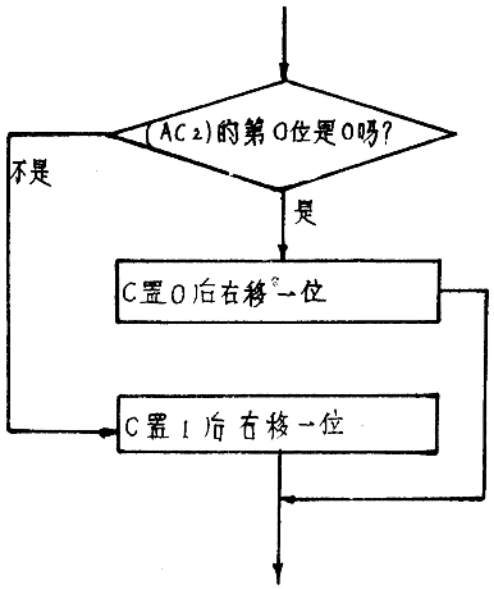


图6

因此其程序为：

MOVL# 2，2，SZC；判断第0位是0码?

MOVOR 2，2，SKP；不是0，C置1除2並跳到下一条

MOVZR 2，2；是0，C置0除以2

4、加1 INC(INCrement)

将ACS的内容加1后放到移位器里並依次完成辅助操作。在不出现“#”号的条件下，完成

(ACS)+1—→ACD

5、加反 ADC(ADdComplement)

将ACS的内容取反后加ACD的内容，其结果送到移位器，並逐项完成辅助操作。

在不出现“#”号的条件下，完成